Randomized Jumplists With Several Jump Pointers

Elisabeth Neumann

Bachelor Thesis

Randomized Jumplists With Several Jump Pointers

submitted by

Elisabeth Neumann

31. March 2015

TU Kaiserslautern
Fachbereich Informatik

Supervisor: Prof. Dr. Markus E. Nebel

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Alle wörtlich oder sinngemäß übernommenen Zitate sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Kaiserslautern, den 31. March 2015

Elisabeth Neumann

Contents

List of Figures

1. Introduction

Since a long period of time, storing and maintaining information and making it accessible with ease, has been and still is one of the major tasks of computer science. Such data structures are called *dictionaries* and they provide operations such as insertion, deletion and search. Since the introduction of AVL-trees, data structures that provide these operations in $\mathcal{O}(\log(n))$ time, with n the length of the list, are known [7]. Other data structures that also grant these operations in logarithmic time are for example randomized or balanced binary search trees (BST), splay trees, skip-lists and jumplists [8]. The latter are the subject of this bachelor thesis.

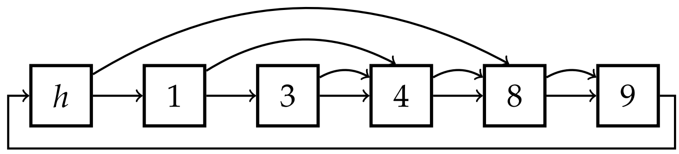


Figure 1.1.: Example of a jumplist of length 5, as defined in [1].

Jumplists are based on a very simple data structure: the circularly closed (singly) linked list. But to avoid long walks along the next-pointers to nodes located near the end of the list, a jump-pointer is added to every node to provide shortcuts to parts further ahead in the list.

Jumplists were introduced by Brönnimann, Cazals and Durand [1] in 2003. The structure of the list, i. e. the targets of the different jump-pointers are independent of the keys in the list as the targets are randomly chosen to form a nested pointer structure. This randomized structure allows easier maintenance and search, resulting in $\mathcal{O}(\log(n))$ expected time complexity for these operations. To facilitate the operations, a header-node containing no key is introduced before the actual list and the last node points to this header.

A deterministic version of jumplists was presented by Elmasry [6]. Similar to weight-balanced trees, a balancing condition for every node is introduced. This condition is fulfilled if the ratio of the number of nodes accessible by the next-pointer and the ones accessible by the jump-pointer is between α and $1 - \alpha$ (included) for $0 < \alpha < \frac{1}{2}$. If however the condition is infringed for a set of nodes, by repeated deletion or insertion, the smallest part of the list containing all these nodes, is rebuilt by choosing for every pointer the best possible target (considering the nested pointer structure and the best partition of the partial list). The deterministic jumplists provide the three basic operations in $\mathcal{O}(\log(n))$ amortized time.

Jumplists are very similar to binary trees: the performance of the search operation in jumplists is linked to the form of the corresponding binary tree, as the necessary number of pointer to reach a node corresponds to the depth of the same node in the binary tree [4]. By cutting off superfluous pointers, the ones that are not part of a shortest path to a node, in the jumplist and "letting it dangle" from the header, a corresponding binary tree is built.

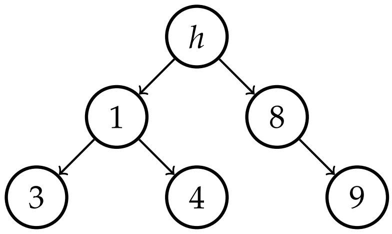


Figure 1.2.: Binary tree corresponding to the jumplist in Figure 1.1.

Another related data structure is the skip-list. Similar to the jump-lists, skip-lists also use additional pointers to speed up the search. These pointers are ordered in towers with the higher located pointers skipping more nodes than the pointers located below. For each layer in the tower, the node has one jump-pointer leaving and one entering. Contrary to jump-lists, a node can have more than one jump-pointer, the number can even vary from node to node.

The search is started at the highest possible layer in the header. If a link of a node in this layer reaches too far, the layer just below in the same tower is tried.

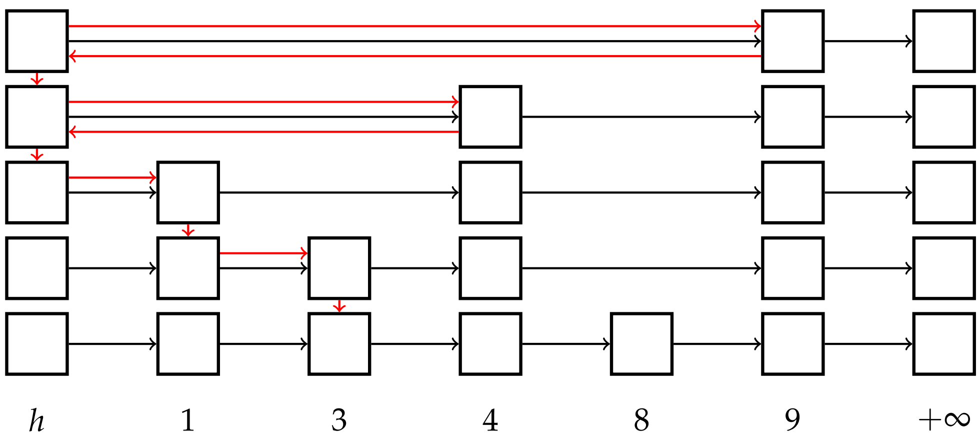


Figure 1.3.: Example of a skip-list. The search path for node 3 is colored in red.

Skip-lifts [9] are another data structure which are related to jumplists. To save space, only at most the two highest layers per tower (except for the header) are preserved, leaving every node, except the header, with exactly 2 jump-pointer. These remaining layers need to be doubly linked, to ensure that every node can be reached.

The search starts at the highest possible layer of the header. Then the right links are followed until the element is found or the link reaches too far. If the layer below is not empty, the search tries to use the element below. If the current element does not have a pointer in the layer below, the algorithm follows the left pointers until an element is found whose layer below is not empty. This process is iterated until the searched element is found.

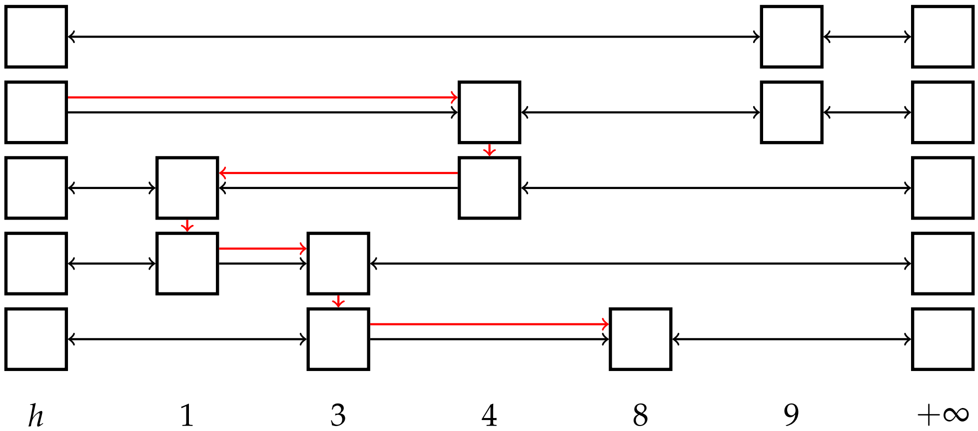


Figure 1.4.: Example of the skip-lift corresponding to the skip-list in Figure 1.3. The search path for node 8 is colored in red.

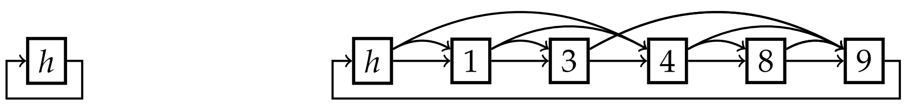
In this bachelor thesis, a slightly modified form of the jumplists will be analysed: the 2-jumplist. Instead of having only two pointers per node, the next- and the jump-pointer, every node will have three: the next-, jump- and jump2-pointer with the latter reaching even more far ahead than the jump-pointer. The thesis includes a presentation of the algorithms for generation, insertion and search as well as proofs for their correctness. Furthermore, expected search costs will be analysed in detail and a time complexity analysis will be conducted for generation and insertion.

2. Preliminaries

The 2-jumplist is the same at its core as the simple jumplist defined by Brönnimann, Cazals and Durand in [1]. Nevertheless, I provide a brief definition of the 2-jumplist and present some more restricted variants of the 2-jumplist. I only consider the randomized jump-list version, as a deterministic version is more difficult to maintain upon insertion and deletion.

A 2-jumplist is a linked list with a header of ordered nodes, according to an ordering $\prec$ on the labels. Every node has a jump and a jump2-pointer, in addition to a next-pointer. The next-pointers form a circularly-closed singly-linked list. The jump- and jump2-pointer of a node h point to nodes a and b respectively picked uniformly at random from the successors of h. This uniform distribution needs to be preserved and thus needs to be restored upon insertion or deletion. For a and b holds that a.label $\preceq$ b.label (x.label is the label of node x).

To this basic definition, a few constraints are added to generate a set of more restricted list-families. These families are defined incrementally, meaning that a constraint introduced in a family above also applies to the families below.



(a) empty 2-jumplist (b) example of a non-empty 2-jumplist

Figure 2.1.: Examples of the non-restricted 2-jumplist.

1. **No crossing edges:** This restriction requires that for the order $\prec$ on the labels and for the jump- and jump2-pointer, the following must hold:

   $$(\nexists xy)(x.label \prec y.label \prec x.jump.label \prec y.jump2.label) \textbf{ (1)}$$
   and
   $$(\nexists xy)(x.label \prec x.jump.label \prec y.label \prec x.jump2.label \prec y.jump2.label) \textbf{ (2)}$$

   x.jump resp. x.jump2 refer to the target-node of x's jump- resp. jump2-pointer. **(a)** and **(b)** equal to the forbidden configuration depicted in Figure 2.2.

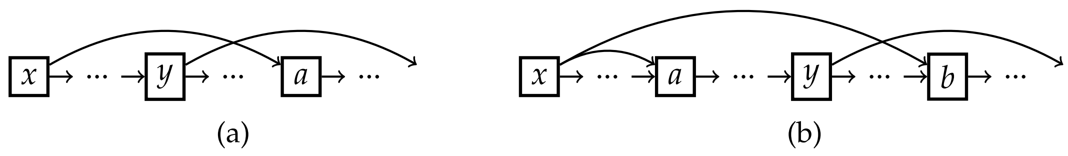


Figure 2.2.: Forbidden configurations.

This restriction results in a nested pointer structure. The list is build from the front backwards and the jump- and jump2-pointer of a node h are not chosen among all the successors of h, but from the partial list starting behind h and ending at (including) the first node, which is the target of a pointer originating from a node located before h (or at the end of the whole list if no such node exists). h is then called header of this partial list, the node ending the list *end* and the partial list itself is referred to as sublist of h. The length of the sublist is called n.

Similarly, I refer to the following partial lists:

next-list: the partial list starting at h.next and ending but not including h.jump. The size of this list is called *nsize*.

jump-list: the list starting at h.jump and ending but not including h.jump2. The size of this list is called *jsize*.

jump2-list: and the list starting at h.jump2 and ending at *end*. The size of this list is called *j2size*.

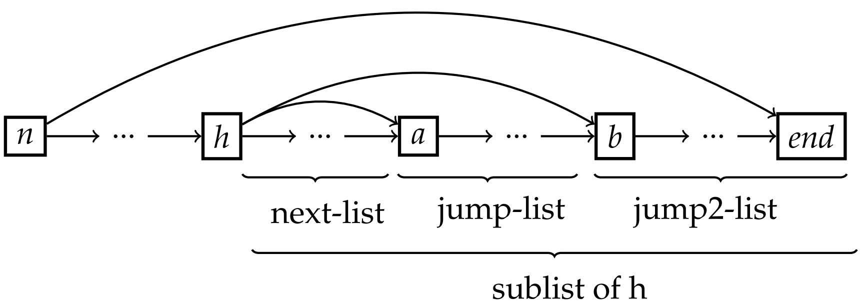
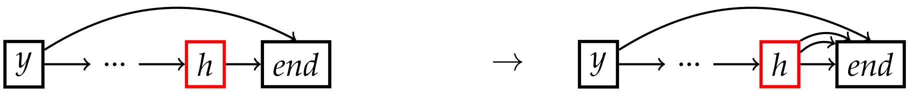


Figure 2.3.: Overview of the defined notations. Node n is the closest predecessor of h with (at least) one pointer reaching over it and *end* is the target of the closest of these/this pointer(s) to h. If no such node n exists, *end* is the last node of the list. *end* thus limits the sublist of h.

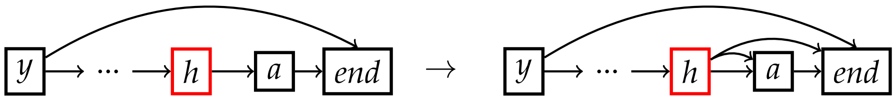
2. **The jump- and jump2-pointer of every node point to different nodes, if it is possible:** This restriction requires that if the sublist of node h contains at least two nodes, the jump- and the jump2-pointer must point to different nodes. This

restriction contributes to the quality of the list as search-data-structure, as pointers to the same node are a waste of potential.

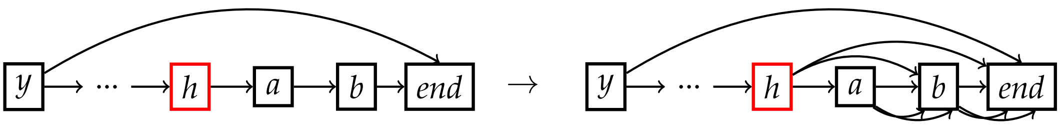
3. **The jump-pointer does not point to the node after it's origin, if possible:** Like the restriction before, this can only be assured if the sublist has an appropriate length. To ensure both restrictions, a sublist of length at least three is needed, for length one, two and three, the special cases displayed in 2.4 are needed. Correspondingly, I refer to the sublist of a node without successor as target-sublist.



(a) length of sublist=1:
nsize=jsize=0, j2size=1.



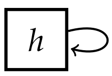
(b) length of sublist=2:
nsize=0, jsize=j2size=1.



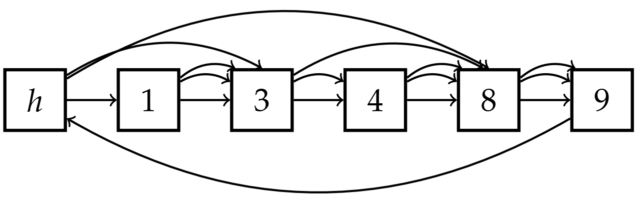
(c) length of sublist=3:
nsize=jsize=j2size=1.

Figure 2.4.: Special cases in the 2-jumplist.

As for the list family before, this restriction makes searching faster, as the jump-pointer is not wasted on a node already referenced by the next-pointer.



(a) empty 2-jumplist



(b) example of a non-empty 2-jumplist

Figure 2.5.: Examples of the (restricted) 2-jumplist.

In this thesis, only the third list-family will be covered, as it is the most efficient one according to search-costs. The restrictions 1-3 will be called **conditions of correctness**, as they will be used in the proofs of correctness.

3. Algorithms

For this thesis, I wrote algorithms for search, generation and insertion of a node. They are adjusted versions of the algorithms for the original jumplist as described in [1].
In the following sections, these algorithms are described and implementations are given in form of pseudo-code.

In my realization of the 2-jumplist, the header as well as the length of the (whole) list (without the header) are stored for every 2-jumplist under the names *list.header* resp. *list.length*.
Every node contains 6 variables: the label, the next-, jump-, jump2-pointer and the sizes of the next- resp. jump-list (called *nsize* resp. *jsize*).

3.1. Search

Searching in a 2-jumplist is very intuitive. The jump- and jump2-pointers are used as shortcuts unless they point too far, always using the one that points farthest ahead without being greater that the label that we look for.

For the search, the if (insert)-statements in lines 7 and 12 can be ignored, as they are only needed in the insertions algorithm. If a node with the sought label is found, a reference to the target and the boolean true are returned to indicate that the target has been found. Upon unsuccessful search, false is output and a reference to the element which had the largest smaller label compared to the label that is looked for. This reference will be needed in the insertion algorithm.

Algorithm 1 search(int k, boolean insert)

```
node current = list.header;
while (current.next ≠ list.header) do
    if (current.jump2.label ≤ k) then
        current = current.jump2;
    else
        if (current.jump.label ≤ k) then
            if (insert) then
                current.jsize++;
            current = current.jump;
        else
            if (current.next.label ≤ k) then
                if (insert) then
                    current.nsize++;
                current = current.next;
            else
                if (current.label == k) then                ▷ element found
                    return new tuple(current, true);
                else                                        ▷ element not found
                    if (insert) then
                        current.nsize++;
                    return new tuple(current, false);
return new tuple(current, false);       ▷ k is bigger than every label in the list
```

3.2. Construction of the List

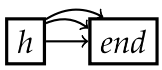
The procedure **Preprocessing** generates a 2-jumplist from sorted list *l*. The data structure of *l* does not matter, only line 6 needs to be adapted to it, which is done by altering the method of iteration trough *l*.

Procedure **Preprocessing** only produces a circularly-closed list, by generating nodes with only the label and the next-pointer specified. The more interesting part of the work is done by the procedure **rebalance**: setting the jump- resp. jump2-pointer and the *nsize* resp. *jsize* fields.

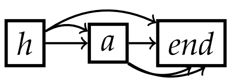
Algorithm 2 Preprocessing(list l)

list.header = new node();
list.length = length(*l*); ▷ length without the header
if (*list.length* $== 0$) **then** ▷ the list is empty
    list.header.next = *list.header*;
else
    /*Produce a circulatory closed linked list with *list.header* as header*/
    node last = **rebalance**(*list.header*, length);
    last.*nsize* = 0;
    last.*jsize* = 0;

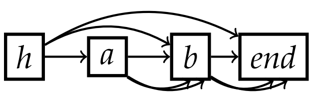
Procedure **rebalance** iterates trough the list, considering each node exactly once. For every node h, the jump- and jump2-pointer are determined, as well as *nsize* and *jsize*. To do so, two random integers m and $m2$ are drawn, which indicate the target-nodes of the jump- resp. jump2-pointer in the target-sublist of h. Choosing integer i corresponds to pointing at the (i)-th node of the sublist of h.
Lines 1-9 take care of the special cases, which are displayed in Figure 3.1. After the target-nodes have been chosen, *nsize* and *jsize* can be computed. As a reference to the target nodes is needed to assign them to the pointers, it is necessary to navigate through the list. This navigation is used, to assign the missing fields to the nodes in the next- jump and jump2-lists of h (recursive calls in lines 14-16), making random access to nodes unnecessary. It is possible that the third recursive call (line 17) is made with *end* as header (it happens if $m2 = n$ is chosen). In this case, no pointers need to be assigned and only the last node, *end* needs to be returned.



(a) Special case for $n = 1$. It is handled in **rebalance**, line 2.



(b) Special case for $n = 2$. It is handled in **rebalance**, line 5.



(c) Special case for $n = 3$. It is handled in **rebalance**, line 8.

Figure 3.1.: Special cases in **rebalance**.

Algorithm 3 rebalance(node h, int n)

```
node end = h;                                          ▷ if n==0, return h
if (n == 1) then
    /*Adjust pointers and sizes according to Figure 3.1 (a)*/
    node end = h.jump2;
if (n == 2) then
    /*Adjust pointers and sizes according to Figure 3.1 (b)*/
    node end = h.jump2;
if (n == 3) then
    /*Adjust pointers and sizes according to Figure 3.1 (c)*/
    node end = h.jump2;
if (n > 3) then
    /*Chose m, m2 with (2 ≤ m < m2 ≤ n) uniformly at random.*/
    h.nsize = m − 1;
    h.jsize = m2 − m;
    h.jump = rebalance(h.next, m − 1);
    h.jump2 = rebalance(h.jump, m2 − m);
    node end = rebalance(h.jump2, n − m2);
return end;
```

3.3. Insertion

Upon insertion, the uniform distribution of the pointers needs to be restored. Of course, a call of **rebalance** on the list with the new element would solve the problem, resulting in a linear time complexity. The algorithm presented below however has a logarithmic time complexity, significantly improving the naive insertion algorithm. It does not completely get along without **rebalance**, but it rarely resorts to it, if it is impossible to restore uniformity otherwise.

As the insertion is more complex than the search and the generation, I present a brief overview over the used procedures and their functions:

- **Insert:**
  - unsuccessful search.
  - insert *new* into the linear list.
- **restore_uniformity and relocate_pointer:**
  - restore uniformity of the nodes located before the direct predecessor of *new*.

- **usurp :**
  - restore uniformity of the direct predecessor of *new* and the following nodes.

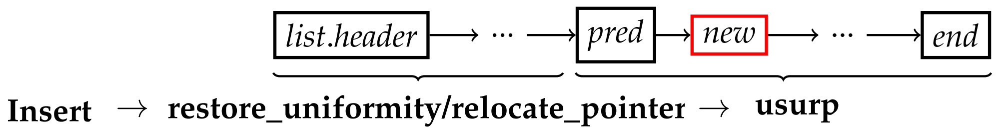


Figure 3.2.: Process flow of procedures used during insertion.

The preprocessing is executed by **Insert**: To determine where the node with the new label needs to be inserted, an unsuccessful **Search** is carried out. This search is also used to update the *nsize* and *jsize* fields of the nodes before the insertion position (lines 7 and 12 in **Search**). If the jump-pointer of node *current* is followed during the search, the new node must be in the jump-list of *current* and *jsize* needs to be increased by one. On the other hand, if the next-pointer is followed, *new* must be in *current*'s next-list, resulting in an increase of *nsize* of *current*. Finally a node with the new label is inserted at the right place in the list and **restore_uniformity** is called to restore the uniform distribution of the pointers.

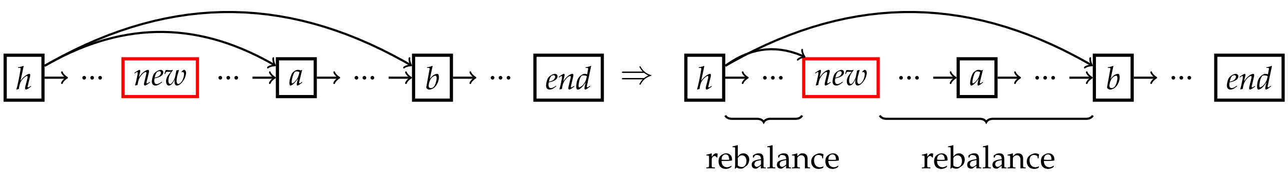
Algorithm 4 Insert(int new)

```
search(new, true);
list.length++;
node new = new node();
/*Insert new in the linked list*/
restore_uniformity(list.header, new, list.length, null)
```

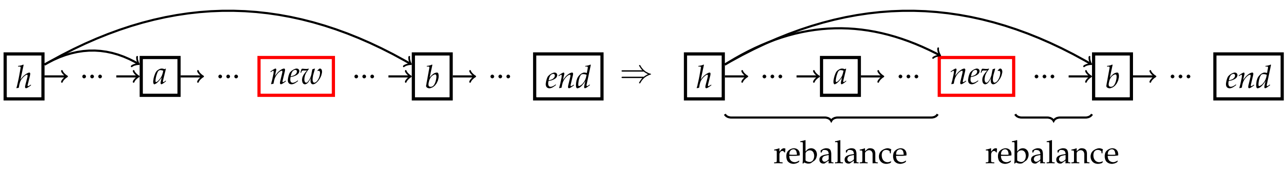
For the nodes h which are located before the immediate predecessor of *new* and have *new* in their sublist, the new node is a possible target for their pointers. Therefore, the pointers of these nodes h are not uniformly distributed anymore. The uniformity is restored by **restore_uniformity** and **relocate_pointer**. **restore_uniformity** directs **relocate_pointer** by determining in which part of the sublist the new node is located and **relocate_pointer** does the actual relocation of pointers.

Algorithm 5 restore_uniformity(node h, node new, int n, node end)

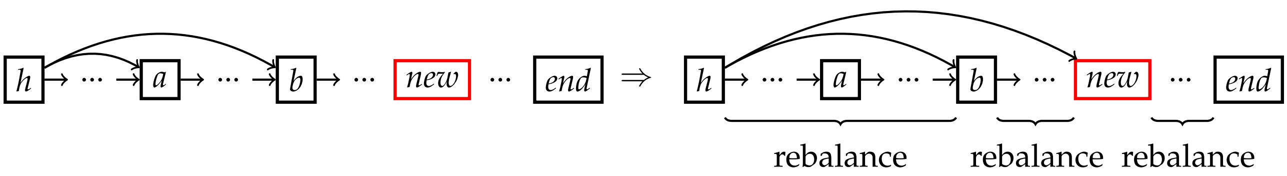
```
if (h.jump == null) then                                  ▷ new is the last node in the list
    h.jump = h.jump2 = new;
    new.nsize = new.jsize = 0;
else
    if (new.label > h.jump2.label) then                   ▷ new in jump2-list
        relocate_pointer(h, new, n, "jump2-list", end);
    else
        if (new.label > h.jump.label) then                ▷ new in jump-list
            if (h.nsize == 0) then                        ▷ Special case
                /* adjust pointers and sizes according to Figure 3.5 (c) */
            else
                relocate_pointer(h, new, n, "jump-list", end);
        else
            if (new.label > h.next.label) then            ▷ new in next-list
                relocate_pointer(h, new, n, "next-list", end);
            else                                          ▷ new directly behind h
                ursup(h, true, end, n);
```



(a) list = next-list in **relocate_pointer**.

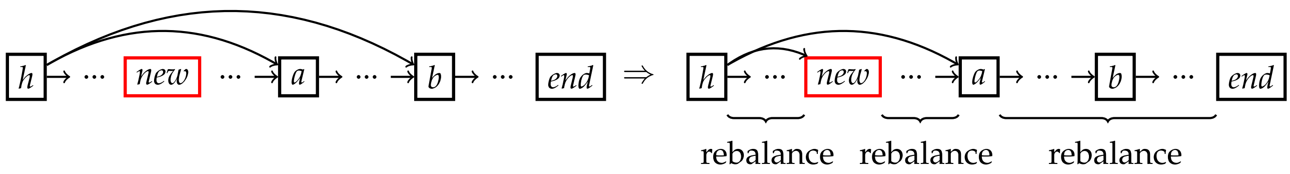


(b) list = jump-list in **relocate_pointer**.

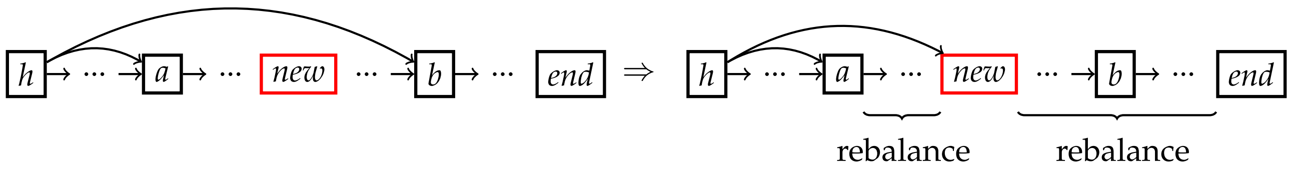


(c) list = jump2-list in **relocate_pointer**. The names of the pointers need to be switched

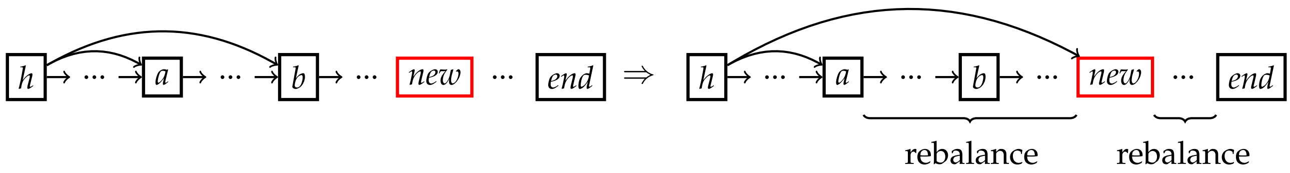
Figure 3.3.: Calls of **rebalance** in **relocate_pointer**, if the jump-pointer is bent.



(a) list = next-list in **relocate_pointer**. The names of the pointers need to be switched.



(b) list = jump-list in **relocate_pointer**.



(c) list = jump2-list in **relocate_pointer**

Figure 3.4.: Calls of **rebalance** in **relocate_pointer**, if the jump2-pointer is bent.

Algorithm 6 relocate_pointer(node h, node new, int n, String list, node end)

boolean bend = True with probability $\frac{2}{n-1}$;
if (bend) **then** ▷ one pointer will be bent
  boolean bend_jump = True with probability $\frac{1}{2}$;
  if (bend_jump) **then**
    h.jump=new;
    /*Adjust sizes and call **rebalance**
      according to Figure 3.3*/
  else ▷ bend jump2-pointer
    h.jump2=new;
    /*Adjust pointers, sizes and call **rebalance**
      according to Figure 3.4*/
else ▷ no pointer will be bent
  if (list == "jump2-list") **then**
    restore_uniformity(h.jump2, new, $n - h.nsize - h.jsize - 1$, end);
  if (list == "jump-list") **then**
    restore_uniformity(h.jump, new, $h.jsize$,h.jump2);
  if (list == "next-list") **then**
    restore_uniformity(h.next, new, $n.jsize$, h.jump);

Starting at *list.header*, all the nodes *h*, which are located before the immediate predecessor of *new* and have *new* in their sublist, need to be processed.

If *h* is considered as header by **restore_uniformity** and it was the last node in the list before the insertion, *new* is the last node in the list after the insertion. *h*s jump- and jump2.pointer are still null at this point and thus are set to *new* in the special case (line 1 in **restore_uniformity**), which handles this situation. *nsize* of *h* has already been updated during the **Search** and the sizes of *new* are updated in line 3 in **restore_uniformity**.

If the sublist of *h* is at most three, one of the special cases in Figure 3.5 applies. Only one of them, (c), concerns **restore_uniformity/relocate_pointer** and is handled at line 9. The others are managed by **usurp**.

If on the other hand n of *h* is at least four, other measures need to be taken. Both the jump- and jump2-pointer can now point to *new*, which happens with probability $\frac{1}{n-1}$ (the successor of *h* is no possible target) for both. Therefore, the algorithm bends a pointer of *h* with probability $\frac{2}{n-1}$ and chooses it randomly, restoring the uniformity of the pointer distribution of *h*.

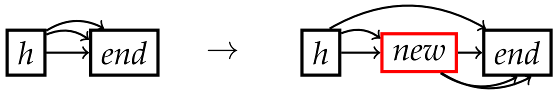
If a pointer has been bent, it is quite difficult to restore the uniformity of the nodes in the sublist of *h*, as their sublists have changed. Therefore, **rebalance** is called on determined parts of the sublist to restore the uniform distribution of the pointers. The parts which must be restored, depend on the position of *new* relative to *h* and on the pointer that has been bent. The different cases are shown in Figures 3.3 and 3.4. After the calls of **rebalance**, the uniformity of the pointers of all the nodes has been restored and the insertion is terminated.

If the algorithm chooses not to bend, **restore_uniformity** is called recursively. Which node will be considered next, depends again on the position of *new* relative to *h*. If *new* is in the next-list of *h*, **restore_uniformity** has to be called on *h*.next as it has *new* in its sublist. But if *new* is in the jump- or jump2-list of *h*, the nodes of the next- resp. of the next- and jump-list of *h* do not contain *new* in their sublist and therefore do not need to be considered. In these cases, **restore_uniformity** is called on *h*.jump resp. *h*.jump2.

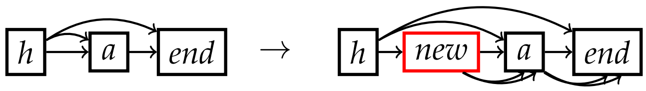
If the execution of **restore_uniformity** is not stopped by a call of **rebalance**, the algorithm will eventually consider the predecessor of *new* as header and **usurp** needs to be called.

Procedure **usurp** can be separated into two phases.

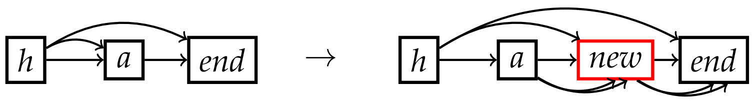
In the first phase, only the immediate predecessor of *new*, *pred* is processed. Due to the insertion of *new*, the node *next* that directly followed *pred* before the insertion is now a possible target for a pointer of *pred*. If the sublist of *pred* has size at most three, special cases (a) or (b) in Figure 3.5 arise, which also determine the pointers of *new* and the insertion is finished. But if the sublist has a length greater than four, the jump- or



(a) Special case n = 2. It is handled in **usurp**, line 1.



(b) Special case n = 3. It is handled in **usurp**, line 4.



(c) Special case n = 3. It is handled in **restore_uniformity** line 9.

Figure 3.5.: Special cases during insertion. Bending of the pointers occur with probability 1.

jump2-pointer can target *next* with probability $\frac{1}{n-1}$ (*new*, the direct successor, is obviously no target for the pointers of *pred*) each. Therefore, one randomly chosen pointer of *pred* is bent with probability $\frac{2}{n-1}$. As in **restore_uniformity/relocate_pointer**, if a pointer is bent, **rebalance** needs to be called.

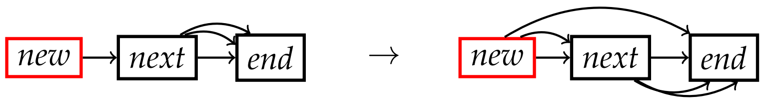
Otherwise, **usurp** is called recursively, entering the second phase, and the node *new* is taken into consideration. *new* had obviously no jump- and jump2-pointer until now. To avoid a call of **rebalance**, the pointers need to be determined otherwise.

If ($n \leq 3$) for *new*, the pointers are simply determined by the special cases in Figure 3.6.

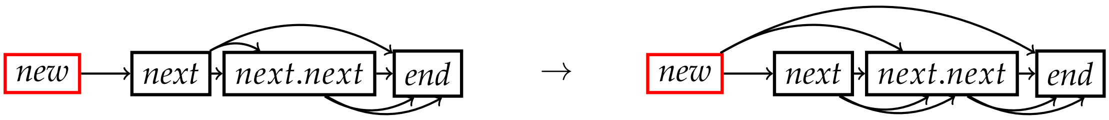
Otherwise, the pointers of *next*, the node following *new* can be stolen. Since *next* possesses pointers with nearly the requested distribution. Only the direct successor of *next* was no target for the pointers of *next*, but is for the pointers of *new*. Therefore, the algorithm steals the pointers of *next* and sets one randomly chosen pointer of *new* to *next*.next with probability $\frac{2}{n-1}$. If a pointer is bent, **rebalance** has to be called according to Figures 3.7 and 3.8 and the algorithm can terminate.

If, however, the algorithm chooses to steal the unchanged pointers of *new*, **usurp** is recursively called on *next*, which now has no valid pointers anymore.

The recursive calls of **usurp** come to an end either by a call of **rebalance** or one of the special cases. Termination is guaranteed as the fourth-to-last node of the list can only have a sublist containing at most three nodes, leading to special case (b) in Figure 3.6.

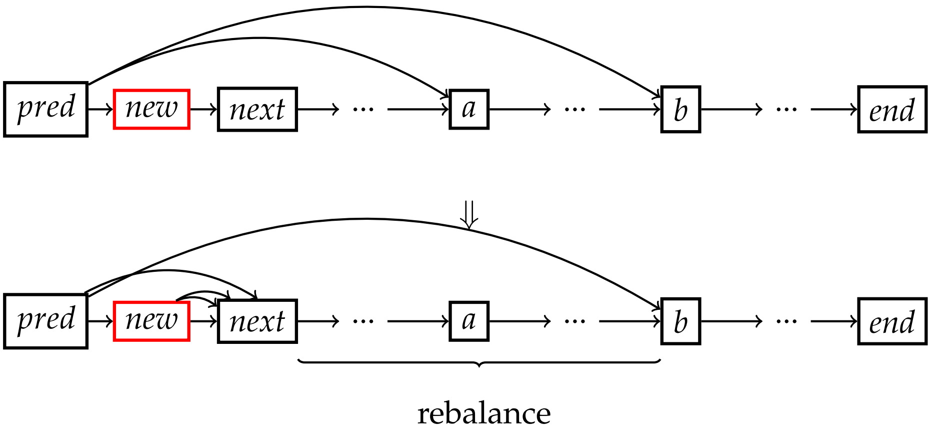


(a) Special case: n=2. It is handled in **usurp** line 1.

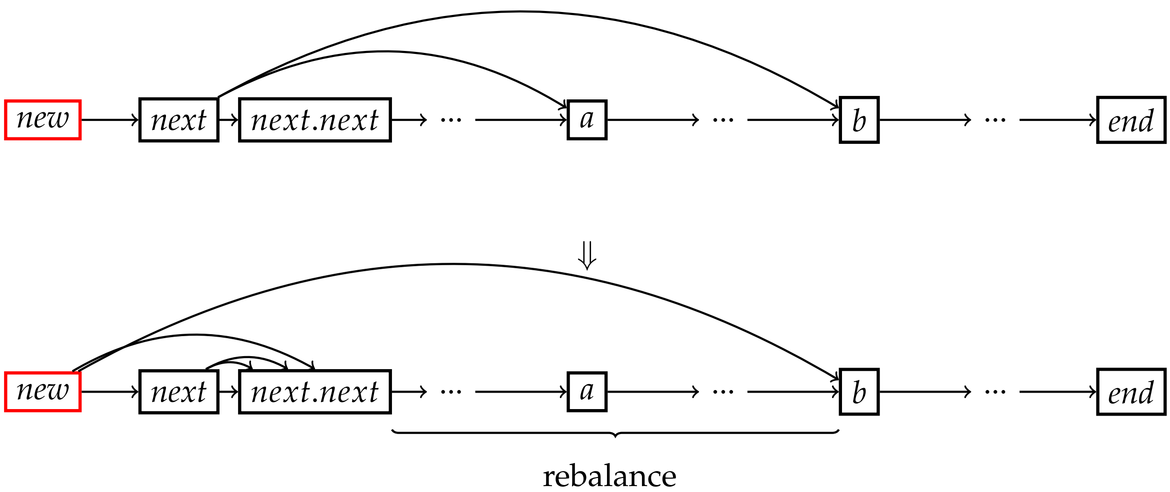


(b) Special case: n=3. It is handled in **usurp** line 4.

Figure 3.6.: Special cases in **usurp**. The bending of the pointers occur with probability 1. *new* represents the new node or any node whose pointers have been stolen in the previous iteration.

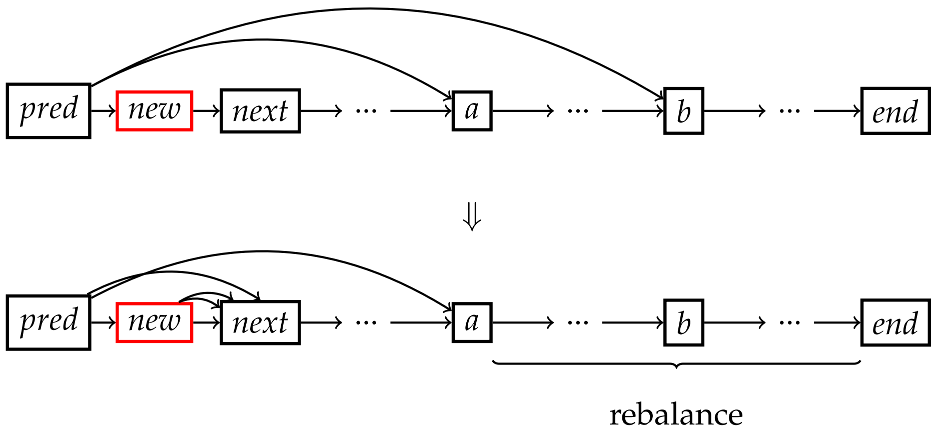


(a) (first == true) in **usurp**.

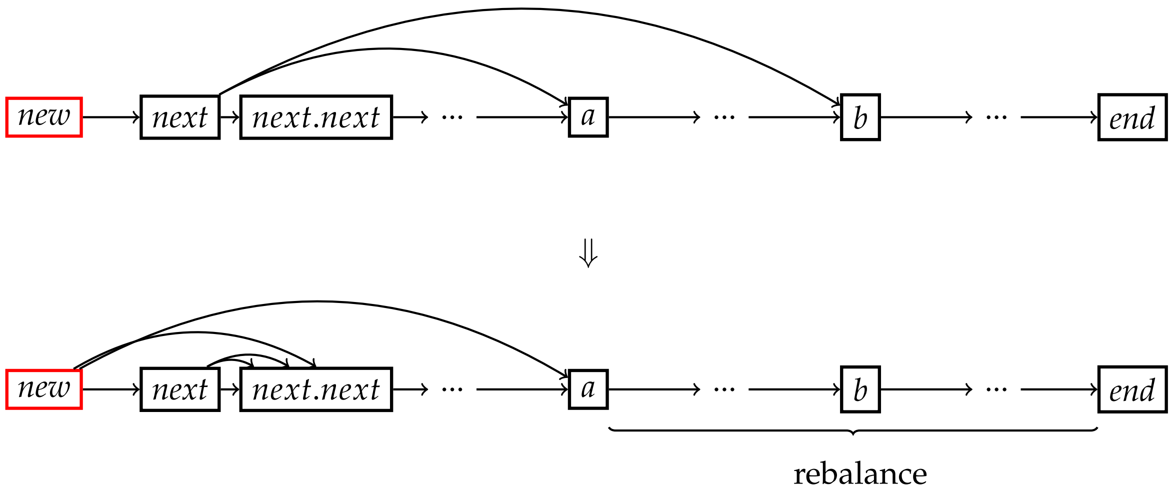


(b) (first == false) in **usurp**. Note that this figure is valid for every recursion in the second phase. Only in the first recursion *new* is really the inserted node. In the following recursions, *new* represents the node whose pointers have been stolen in the previous recursive call.

Figure 3.7.: Calls of rebalance in usurp, if the jump-pointer is bent.



(a) (first == true) in **usurp**



(b) (first == false) in **usurp**. Note that this figure is valid for every recursion in the second phase. Only in the first recursion *new* is really the inserted node. In the following recursions, *new* represents the node whose pointers have been stolen in the recursion before.

Figure 3.8.: Calls of rebalance in usurp, if the jump2-pointer is bent.

Algorithm 7 usurp(node h, boolean first, node end, int n)

```
if (n == 2) then
    /*Adjust pointers and sizes according to Figure 3.6 (a) */
else
    if (n == 3) then
        /*Adjust pointers and sizes according to Figure 3.6 (b) */
    else
        boolean bend = True with probability 2/(n-1);
        if (bend) then                                  ▷ a pointer will be bent
            boolean bend_jump = True with probability 1/2;
            if (bend_jump) then                         ▷ bend the jump pointer
                */Adjust pointers, sizes and call rebalance
                according to Figure 3.7/*
            else                                        ▷ bend the jump2-pointer
                */Adjust pointers, sizes and call rebalance
                according to Figure 3.8/*
        else                                            ▷ no pointer will be bent
            if (!first) then
                /*Steal pointers from next and adjust sizes*/
            usurp(h.next, false, h.jump, h.nsize);
```

3.4. Correctness

The correctness of the generation and the insertion procedures will be elaborated, whereas the search will be omitted, due to its trivial nature.

3.4.1. Generation

As the main part of the work, the setting of the jump- and jump2-pointer, is provided by the procedure **rebalance**, a detailed proof of its correctness will be shown. The rest of the generation, the initialization of the nodes and the generation of the linked list, will be omitted again due to triviality.

To prove the correctness of **rebalance**, I show that after the execution, every node fulfils the following **conditions of correctness**, which have already been established in Preliminaries:

- **(c1)**: The jump and jump2- pointers do not cross any other pointer in the list.
- **(c2):** If $n \geq 4$:
  - **(c2.1):** Its pointers are chosen uniformly at random from the sublist of the node.
  - **(c2.2):** The pointers do not point to the same node.
  - **(c2.3):** No pointer points to the node directly after its origin.

  (c2'): Otherwise ($1 \leq n < 4$), the pointers are arranged according to Figure 3.5.

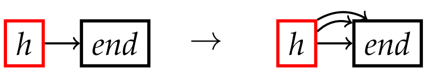
To this end, I prove a more general theorem concerning **rebalance**. Correctness then directly follows from this theorem.

Theorem 1. *After the call of* ***rebalance****(h, n), all nodes in the partial list starting at h and ending at (not including) end, fulfil the conditions of correctness and the call returns end.*

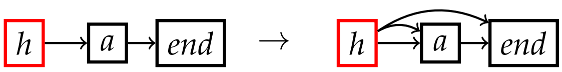
Proof. The proof is conducted by induction over n.

Induction Beginning:

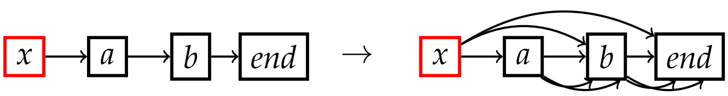
If $n = 0$, only $end = h$ needs to be returned. This is done in line 1.



(a) $n = 1$.



(b) $n = 2$.



(c) $n = 3$.

Figure 3.9.: Special cases in **rebalance**.

For the other three cases ($n = 1, 2, 3$), the conditions of correctness need to be verified:

- **(c1):** In these three cases, no pointers from outside of the considered partial list (from h to end, each included) can point to any other node than end, by definition of end. Therefore, it is sufficient to show that no pointers cross in the considered partial list, which is obvious from Figure 3.9.

- **(c2′):** (**(c2)** does not apply.)
  See lines 2, 5 resp. 8 for ($n = 1$), ($n = 2$) resp. ($n = 3$).

Furthermore, *end* needs to be returned. In lines 4, 7 resp. 10, *end* is set to the rightmost node and returned in line 18.

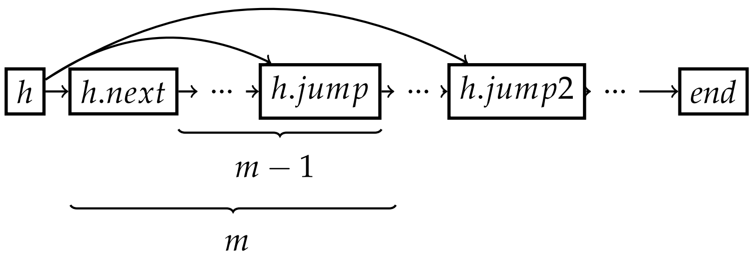
Induction Hypothesis:

For an arbitrary but fixed number n and a node h with a sublist of length $l < n$, holds that after the call of rebalance(h, l), all nodes in the partial list starting at h and ending at (not including) *end*, fulfil the conditions of correctness and the call returns *end*.

Induction Step:

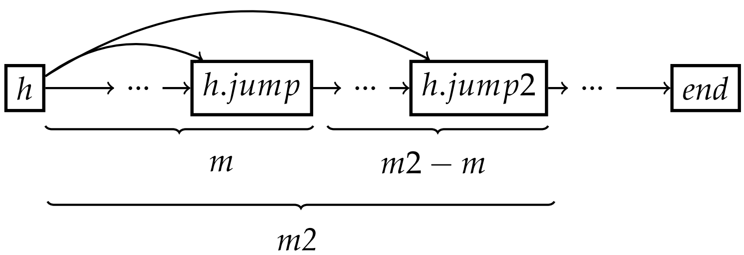
From the **Induction Hypothesis** follows that for a node x and natural number l with $2 \leq l < n$, **rebalance**(x, l) returns the last node (the l-th one) from the sublist of x. Therefore, it follows for m and $m2$ with ($2 \leq m < m2 \leq n$) that:

- **rebalance**(h.next, $m - 1$) returns the ($m - 1$)-th node in the sublist of h.next, which is the m-th node in the sublist of h.

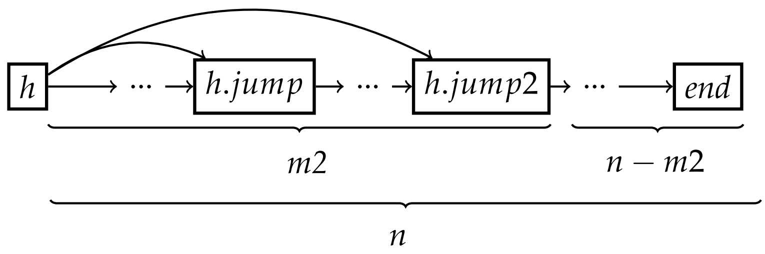
  Thus, h.jump must be the m-th node in the sublist of h.

- **rebalance**(h.jump, $m2 - m$) returns the ($m2 - m$)-th node in the sublist of h.next, which is the $m2$-th node in the sublist of h.

  Thus, h.jump2 must be the $m2$-th node in the sublist of h.

- **rebalance**(h.jump2, $n - m2$) returns the ($n - m2$)-th node in the sublist of h.next, which is the n-th node in the sublist of h.



Thus, *end* must be the last node in the sublist of h.

The conditions of correctness only need to be verified for h. Because the pointers of the other nodes in the partial list from h to *end* (not included) are set by the three recursive calls in lines 15, 16 and 17 and not changed afterwords, the conditions of correctness for these nodes are ensured by the **Induction Hypothesis**.

- **(c1):** By the choice of m and $m2$, only nodes contained in the sublist of h can be targeted by its pointers. From outside of the considered partial list, pointers cannot point to any other node than *end*, by definition of *end*. This results in a nested pointer structure without crossings.
- **(c2):** ((c2)′ does not apply)
  - **(c2.1):** As the pointers are directly determined by m and $m2$, which are chosen uniformly at random with $2 \leq m < m2 \leq n$, the statement follows.
  - **(c2.2):** This holds as m and $m2$ are chosen s. t. $m < m2$.
  - **(c2.3):** This holds as m and $m2$ are chosen s. t. $2 \leq m, m2$.

From lines 17 and 18 follows that the last node of the sublist of h is returned. □

Corollary 1. *Calling **rebalance**(list.header, list.length) results in a correct 2-jumplist.*

3.4.2. Insertion

The main work for the insertion is provided by three procedures, **restore_uniformity**, **relocate_pointer** and **usurp**, with the first two working closely together. Before the call of these procedures, a simple preprocessing (inserting *new* in the right space in the linked list) has been executed by **Insert**, which does not need to be considered in this proof, due to triviality.
The correctness of the procedures **restore_uniformity/relocate_pointer** and **usurp** will be proven separately and joined at the end to prove correctness of the complete insertion.

rebalance_insert/bend_procedure

Before working on **restore_uniformity/relocate_pointer**, two lemmas will be provided to simplify the following proof.

Lemma 1. *If node new is in the sublist of a node h , then h is not skipped by* ***restore_uniformity/relocate_pointer****.*

Proof. The proof is conducted by contradiction.

Assumption: There exists a node, which is skipped by **restore_uniformity/ relocate_pointer**, but *new* is part of its sublist. Let h be the first of these nodes. This leads to two cases:

Case 1: $h = list.header$. In this case, h must be considered by **restore_uniformity/ relocate_pointer** as **Insert** calls **restore_uniformity** with *list.header* as argument (line 5 in **Insert**).

Case 2: $h \neq list.header$. As h was chosen to be the first node skipped by **restore_uniformity/ relocate_pointer**, one node h' must have been the header of **restore_uniformity/ relocate_pointer** just before skipping h.

To ensure that *new* is in the sublist of h, h must be in next-, jump- or jump2-list together with *new*. But in this case, h would not be skipped.

As both cases lead to contradictions, the assumption must be wrong. □

Lemma 2. *Let h be a node with a sublist of length $n-1 \geq 4$ before insertion (target sublist of length $(n-2)$). If node new is inserted into the sublist, the procedure of bending one pointer of h with probability $\frac{2}{n-1}$ and choosing the pointer at random restores the uniformity of the pointer-distribution of h.*

Proof. Two cases need to be distinguished:

- **before insertion:** one pointer points to a.
  after insertion: one pointer points to a, the other to *new*.

$$\underbrace{\frac{n-3}{\binom{n-2}{2}}}_{\substack{\text{Prob. that one pointer}\\ \text{pointed to } a\\ \text{before the}\\ \text{insertion}}} \cdot \underbrace{\frac{2}{n-1}}_{\substack{\text{Prob. that one}\\ \text{pointer will be bent}}} \cdot \underbrace{\frac{1}{2}}_{\substack{\text{Prob that the right}\\ \text{pointer is chosen}}} = \frac{2(n-3)}{(n-2)(n-3)} \cdot \frac{2}{(n-1)} \cdot \frac{1}{2} = \frac{1}{\binom{n-1}{2}}$$

- **before insertion:** one pointer points to a, the other to b.
  after insertion: one pointer points to a, the other to b.

$$\underbrace{\frac{1}{\binom{n-2}{2}}}_{\substack{\text{Prob. that one pointer}\\ \text{pointed to } a \text{ and}\\ \text{the other}\\ \text{pointed to } b}} \cdot \underbrace{\left(1-\frac{2}{n-1}\right)}_{\substack{\text{Prob. that no}\\ \text{pointer will be bent}}} = \frac{2}{(n-2)(n-3)} \cdot \left(\frac{n-1-2}{n-1}\right) = \frac{1}{\binom{n-1}{2}}$$

□

With these lemmas at hand, I can attend to the correctness of **restore_uniformity/relocate_pointer**.
For the proof, I consider the nodes h_0, h_1,. . . ,h_k, $k \in \mathbb{N}_0$, which are the different header elements of **restore_uniformity/relocate_pointer**. h_0 is the first header considered by these 2 procedures and thus is also the first node of the list, *list.header*. h_k is the last header of them and its processing will lead to one of four cases, which will be examined later.

I will show by induction that if node h_i is processed by **restore_uniformity/relocate_pointer**, all nodes located before h_i in the list fulfil the conditions of correctness.

Theorem 2. *If the nodes h_0, h_1,. . . ,h_k, $k \in \mathbb{N}_0$, are the different values for header in **restore_uniformity/relocate_pointer**, then all nodes before h_k fulfil the conditions of correctness.*

Proof. The proof is conducted by induction.

Induction Beginning: i=0

h_i is the first node of the list. Thus no nodes are located before h_i.

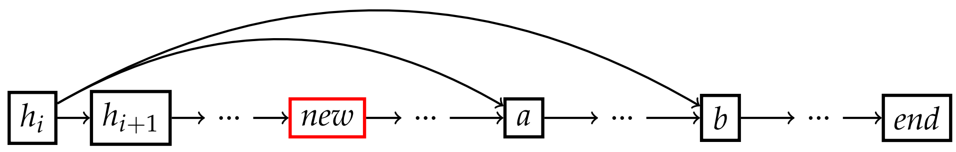
Induction Hypothesis:

For an arbitrary fixed number $i \in \mathbb{N}$, $i < k$ holds that all nodes located before h_i in the list fulfil the conditions of correctness.

Induction Step: $i \to i+1$

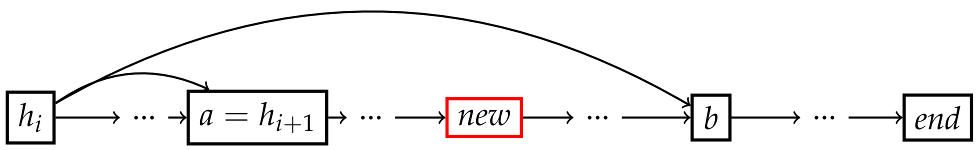
From the **Induction Hypothesis**, follows that all nodes located before h_i fulfil the conditions of correctness.
We need to distinguish 3 cases:

- **Case 1:** h_{i+1} = h_i.next



The conditions of correctness only need to be verified for h_i.

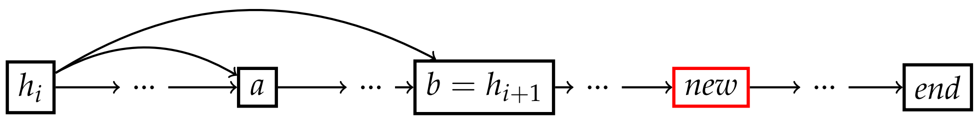
- **Case 2:** h_{i+1} = h_i.jump



Because none of the nodes in the next-list of h_i contain *new* in their sublist, the insertion did not disturb the uniformity of their pointer distribution. Therefore, all these nodes fulfil the conditions of correctness, as they also fulfilled them before the insertion.

Therefore, the conditions of correctness only need to be verified for h_i.

- **Case 3:** h_{i+1} = h_i.jump2



Again, the conditions of correctness only need to be verified for h_i, because none of the nodes in the next- and jump-list of h_i contain *new* in their sublist. The insertion did not disturb the uniformity of their pointer distribution and all these nodes fulfil the conditions of correctness, as they also fulfilled them before the insertion.

In all three cases, the conditions of correctness only need to be verified for h_i:

- **(c1):** As the pointers of h_i are not altered (otherwise h_i would be the last node considered by **restore_uniformity/relocate_pointer**) and every node in the list fulfilled the conditions of correctness before the insertion, the pointers do not cross afterwards.

- **(c2):**
  - **(c2.1):** It needs to be show that every pair of nodes in the target sublist of h_i, including *new* has the same probability to be the target of the pointers of h_i. This is provided by Lemma 2, setting $h = h_i$.
  - **(c2.2):** Because ($n \geq 4$) and *new* is in the sublist of h_i, h_i must have had a sublist of length $\geq$ 3 before the insertion. In this case, the pointers must have pointed to different nodes and as they are not altered also do after the insertion.
  - **(c2.3):** As h_i must have had a sublist of length $\geq$ 3 before the insertion, its pointers did not point to the direct successor before the insertion. The pointers of h_i are not altered and thus this also holds after the insertion.

  (c2'): As h_i is not the last node processed by **restore_uniformity/relocate_pointer**, the sublist of h_i must at least have length 4: A sublist of length one is not possible, because it had a length of at least one before the insertion and thus has length at least two after. Length two and three are excluded as they represent the special cases (a), (b) and (c) in Figure 3.5, which either lead to the end of the insertion or a call of **usurp**.

As Lemma 1 guarantees that no node which has *new* in their sublist is skipped, the Theorem follows. □

Because all the nodes behind the sublist of h_k were not affected by the insertion of *new* (they neither have *new* in their sublist nor was their sublists altered by the algorithm), the conditions of correctness still hold for them after the insertion.

Therefore, it only remains to prove that the conditions of correctness hold for h_k and all the nodes in its sublist.

As mentioned above, four cases can arise when **restore_uniformity/relocate_pointer** is called with h_k as header:

- **h_k is the second-last node in the list.**

  If h_k is the second-last node in the list, it must have been the last node before the insertion. Due to the insertion, *new* is now the last node in the list. Therefore, both pointer of h_k need to be set to *new* and the pointer of *new* to null. This is executed by the special case in line 1 in **restore_uniformity**.

- **$h_k.nsize = 0$.**

  The proceeding for this case is shown in Figure 3.5 (c). Afterwards, all nodes in the sublist of h_k, including h_k, fulfil the conditions of correctness.

- **h_k is directly located before *new*.**

  In this case, **usurp** will be called by **restore_uniformity**. That **usurp** restores the conditions of correctness of h_k and its sublist, will be shown in the section below.

- **A pointer of h_k is bent.**

  Node h_k fulfils the conditions of correctness:

  - **(c1):** As one pointer of h_k is bent, **rebalance** will be called on parts of the sublist of h_k. These parts are chosen s. t. they do not cross the pointers of h_k (see Figures 3.3 and 3.4), thus avoiding any crossings with the pointers of h_k.
  - **(c2):**
    * **(c2.1):** It needs to be shown that every pair of nodes in the target sublist of h_n, including *new*, has the same probability to be the target of the pointers of h_n. This is provided by Lemma 2, setting $h = h_n$.
    * **(c2.2):** Before the insertion, *new* was not contained in the sublist of h_k, thus no pointer of h_k could point to it. If one pointer is set to *new* during the insertion, the two pointers of h_k must point to different nodes.
    * **(c2.3):** Because ($n \geq 4$) and *new* is in the sublist of h_k, h_k must have had a sublist of length ≥ 3 before the insertion. Therefore, its pointers did not point to the direct successor before the insertion. By setting one pointer to *new*, which can also not be the direct successor of h_k (see case 3 above), both pointers do not target the direct successor of h_k.

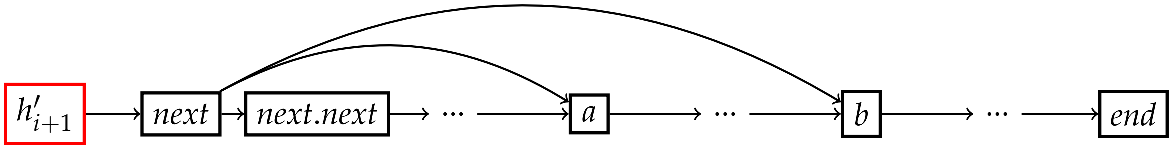
    (c2'): For n must hold that($n \geq 4$), otherwise one of the three special cases above would arise.

  Due to the bending of the pointers of h_k, the sublists of some nodes have changed and the conditions of correctness need to be restored for them. Which nodes are affected is show in Figures 3.3 and 3.4. As **rebalance** is called on exactly these parts, it is clear that all nodes in the sublist of h_k fulfil the conditions of correctness.

usurper

The proof for the correctness of **usurp** is quite similar to the one for **restore_uniformity/ relocate_pointer**.
Consider nodes h'_0, h'_1,…,h'_l, $l \in \mathbb{N}_0$, that are the headers of **usurp**, with h'_0, the direct predecessor of *new* = h'_1. I will show that if node h'_i is considered by **usurp**, it also fulfils the conditions of correctness. As $h'_{i+1} = h'_i$.next, it is sufficient to show the conditions for only one node in each induction step.

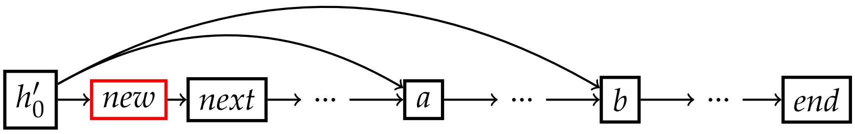
This leads to the following theorem:



Theorem 3. *If h'_0, h'_1,...,h'_l, $l \in \mathbb{N}_0$, are the different values for header in **usurp**, then all nodes between h'_0 (included) and h'_l (not included) fulfil the conditions of correctness.*

Proof. The proof is conducted by induction.

Induction Beginning: i=0



It needs to be shown that h'_0 fulfils the conditions of correctness. But because the arguments are nearly equal to the ones used in the **Induction Step** in the proof of Theorem 2 (the only difference being that *next* is the possible target for h'_0 and not *new*), the verification of the conditions of correctness is omitted here. Refer to the Appendix for the missing part.

Induction Hypothesis

For an arbitrary fixed number $i \in \mathbb{N}, i < (l-1)$ holds that h'_i fulfils the conditions of correctness.

Induction Step: $i \to i+1$

h'_{i+1} fulfils the conditions of correctness:

- **(c1):** As the pointers of h'_{i+1} are not altered (otherwise h'_{i+1} would be the last node considered by **usurp**) and every node in the list fulfilled the conditions of correctness before the insertion, the pointers do not cross afterwards.
- **(c2):**
  - **(c2.1):** For the pointers of *next*, *next.next* was not a possible target. As h'_{i+1} stole the pointers of *next*, *next.next* now needs to be considered as possible target. Therefore, it needs to be shown that every pair of nodes in the target sublist of h'_{i+1}, including *next.next*, has the same probability to be the target of the pointers of h'_{i+1}. This is provided by Lemma 2, setting $h = h'_{i+1}$ and $new = next.next$.

- **(c2.2):** As the sublist of h'_{i+1} has at least length four, the sublist of *next* must have at least length three (the length of sublist of *next* equals the length of sublist of h'_{i+1}-1). Therefore the jump- and jump2-pointer of *next* pointed to different nodes before the insertion and thus do the stolen pointers of h'_{i+1}.
- **(c2.3):** As *next* cannot have pointers on itself, the stolen pointers of h'_{i+1} cannot point to its direct successor *next*.

(c2'): If $(1 \leq n < 4)$, a special case would have been applied and h'_{i+1} would have been the last node considered by **usurp**).

□

Again, all the nodes behind the sublist of h'_l were not affected by the insertion of *new* (they neither have *new* in their sublist nor were their sublist altered by the algorithm. Therefore, the conditions of correctness still hold for them after insertion. Because **usurp** consists of two phases, two cases need to be distinguished: $l = 0$ and $l > 0$. For both of them, it remains to be proven that the conditions of correctness hold for h'_l and its sublist.

usurp can be terminated in two ways:

- **n = 2 or n = 3**.

  The proceeding for this case is shown in Figure 3.6 for $(l > 0)$ and in Figure 3.5 (a) and (b) for $(l = 0)$. Afterwards, h'_l and all the nodes in its sublist, fulfil the conditions of correctness.

- **A pointer of h'_l is bent.**

  Node h'_l fulfils the conditions of correctness:

  - **(c1):** As one pointer of h'_l is bent (if $(l > 0)$ after the pointers have previously been stolen from *next*), **rebalance** will be called on parts of the sublist of h'_l. These parts are chosen s. t. they do not cross the pointers of h'_l (see Figures 3.10 and 3.11), thus avoiding any crossings with the pointers of h'_l. By definition of *end*, pointers from outside of this considered partial list can only point to *end*.
  - **(c2):**
    - **(c2.1):** Due to the insertion of *new* (the theft of the pointers of *next*), *next* (*next.next*) is now a possible target for the pointers of h'_l. Therefore it needs to be shown that every pair of nodes in the target sublist of h'_l, including *next* (*next.next*), has the same probability to be the target of the pointers of h'_l. This is provided by Lemma 2, setting $h = h'_l$ ($h = h'_l$ and *new* = *next.next*).
    - **(c2.2):**

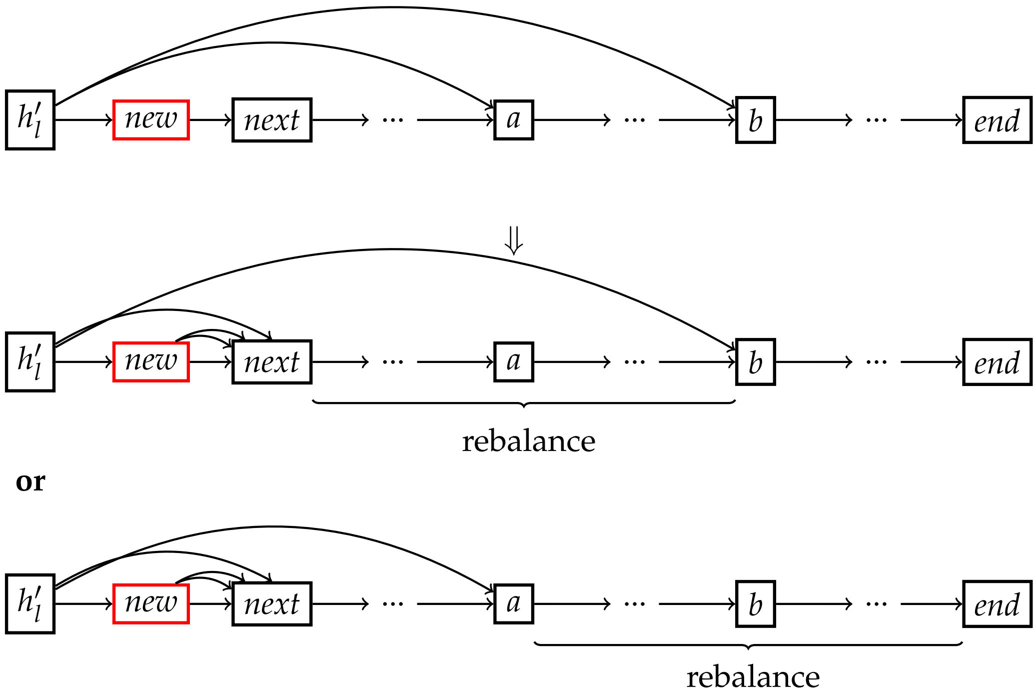


Figure 3.10.: $l = 0$.

- **l=0:** Because $n \geq 4$, h'_l had a sublist of at least length three before the insertion. Therefore, no pointer of h'_l pointed to *next* before the insertion and thus the pointers of h'_l point to different nodes after the insertion.
- **l>0:** Before the insertion, *next* did not point to its direct successor *next.next*, because the length of the sublist of *next* was at least three (length of the sublist of *next* = length of the sublist of h'_l-1). If one pointer is set to *next.next* during the insertion, the two pointers of h'_l must point to different nodes.

* **(c2.3):**
  - **l=0:** As *new* was not contained in the list before the insertion and *new* is the direct successor of h'_l, no pointer of h'_l can point to it after the insertion.
  - **l>0:** As h'_l.next cannot have a pointer on itself, the stolen pointers of h'_l cannot point to its direct successor.

(c2'): For n must hold that ($n \geq 4$), otherwise the special cases above would arise.

Due to the bending of the pointers of h'_l, the sublists of some nodes have changed and the conditions of correctness need to be restored for them. Which nodes are

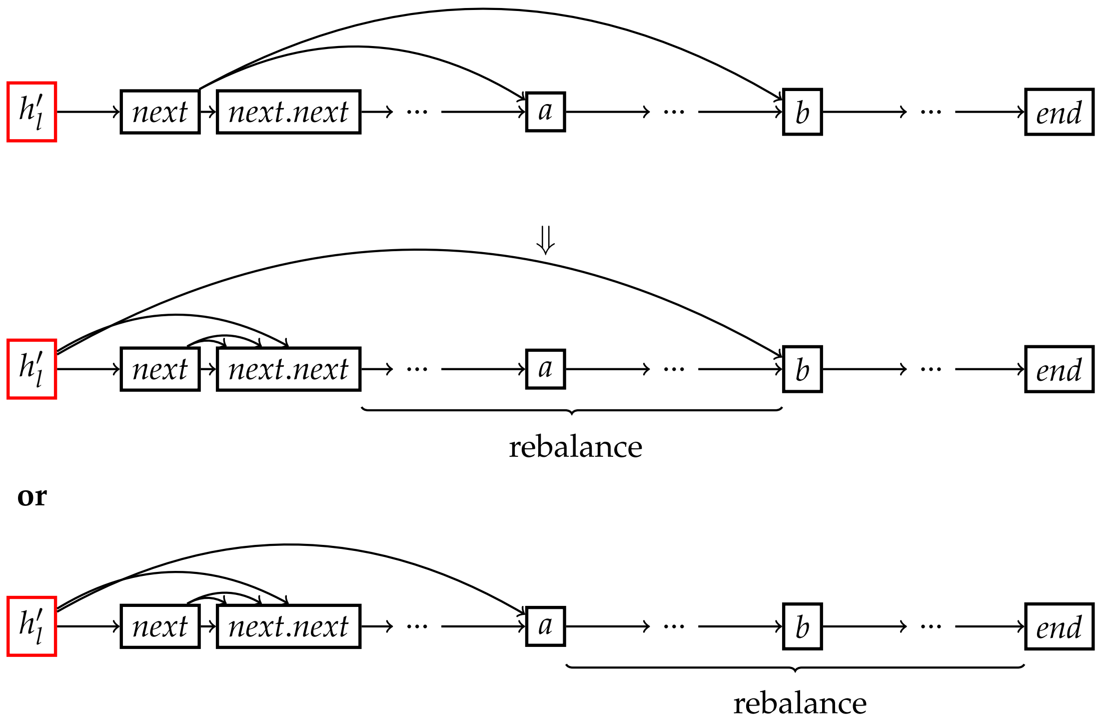


Figure 3.11.: $l > 0$.

affected is show in Figures 3.10 and 3.11. As **rebalance** is called on exactly these parts, it is clear that all nodes in the sublist of h'_l fulfil the conditions of correctness.

Finally, the discussion in the two sections above leads to the following theorem:

Theorem 4. *The insertion of a node in a correct 2-jumplist also results in a correct 2-jumplist.*

4. Analysis of Expected Search Costs

To approximate the expected search costs for a node in the jumplist, two different operations can be counted: the number of pointers that need to be followed or the number of comparisons during the insertion.

As both of these cost models depend on the position of *new*, it is easier to compute the expected costs of a search for *all* nodes in the list. For the number of pointers, this corresponds to the internal path length (IPL) of the list. For the number of comparisons, this quantity will be divided by the length of the jumplist to obtain the expected number of comparisons for one insertion at a random position.

4.1. Expected Internal Path Length of 2-Jumplists

The expected internal path length of a 2-jumplist of length N (header included) is given by the recurrence equation (4.1).

$$C_{\mathrm{IPL}}(1) = 0,\ C_{\mathrm{IPL}}(2) = 1,\ C_{\mathrm{IPL}}(3) = 2,\ C_{\mathrm{IPL}}(4) = 3$$

$$C_{\mathrm{IPL}}(N) = (N-1) + \frac{1}{\binom{N-2}{2}} \sum_{i=1}^{N-3} \sum_{j=1}^{N-2-i} \Big(C_{\mathrm{IPL}}(i) + C_{\mathrm{IPL}}(j) + C_{\mathrm{IPL}}(N-1-i-j) \Big), \qquad N \geq 5. \tag{4.1}$$

The initial condition $C_{\mathrm{IPL}}(1)$ corresponds to an "empty" jumplist, which only consists of the header. In this case, the expected internal path length is zero, as the search starts at the header and no other nodes exist. The initial conditions $C_{\mathrm{IPL}}(2)$, $C_{\mathrm{IPL}}(3)$ and $C_{\mathrm{IPL}}(4)$ correspond to the special cases in Figure 3.1 on page 11.
In the recursive characterization, which holds for $N \geq 5$, the summand $N-1$ represents the contribution to the expected IPL of the pointers of the header. For each node located behind the header, one pointer of the header needs to be followed to access it. The factor $1/\binom{N-2}{2}$ represents the probability for every possible configuration of the pointers of the header, as the target sublist has length $(N-2)$. The double sum

includes every possible size for the next-, jump- and jump2-list of the header, which range from 1 to $(N-3)$.

The representation as double sum can be simplified using symmetry and reverse summation:

$$
\begin{aligned}
&\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}(C_{\mathrm{IPL}}(i)+C_{\mathrm{IPL}}(j)+C_{\mathrm{IPL}}(N-1-i-j))\\
&=\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}C_{\mathrm{IPL}}(i)+\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}C_{\mathrm{IPL}}(j)+\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}C_{\mathrm{IPL}}(N-1-i-j)\\
&=\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}C_{\mathrm{IPL}}(i)+\sum_{j=1}^{N-3}\sum_{i=1}^{N-2-j}C_{\mathrm{IPL}}(j)+\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}C_{\mathrm{IPL}}(j)\\
&=\sum_{i=1}^{N-3}C_{\mathrm{IPL}}(i)\sum_{j=1}^{N-2-i}1+\sum_{j=1}^{N-3}C_{\mathrm{IPL}}(j)\sum_{i=1}^{N-2-j}1+\sum_{j=1}^{N-3}C_{\mathrm{IPL}}(j)\sum_{i=1}^{N-2-j}1\\
&=\sum_{k=1}^{N-3}(N-k-2)C_{\mathrm{IPL}}(k), \qquad N\geq 4.
\end{aligned}
$$

This leads to the following simplified representation of the expected IPL:

$$
\begin{aligned}
&C_{\mathrm{IPL}}(1)=0,\ C_{\mathrm{IPL}}(2)=1,\ C_{\mathrm{IPL}}(3)=2,\ C_{\mathrm{IPL}}(4)=3\\
&C_{\mathrm{IPL}}(n)=N-1+\frac{6}{(N-2)(N-3)}\sum_{k=1}^{N-3}(N-k-2)C_{\mathrm{IPL}}(k), \qquad N\geq 5.
\end{aligned}
\tag{4.2}
$$

Unfortunately, I was not able to find a closed form of $C_{\mathrm{IPL}}(N)$. It does not qualify for a transformation similar to the one described in Section 4.2 of [2], even if the recurrence relation 4.1 of [2] is very similar in shape to $C_{\mathrm{IPL}}(N)$. Due to a different prefactor ($1/\binom{N-2}{2}$ vs. $1/\binom{N}{2}$ in [2]) and boundaries in the sum, the transformation fails at several stages. For the same reasons, the method (in chapter 4.2.2 of [2]) using a multivariate generating function fails at the solution of the differential equation as the resulting equation is not an Euler equation.

Lacking other possibilities, I present two sequences, $\overline{C}_N$ resp. $\underline{C}_N$ which are an upper resp. a lower bounds on $C_{\mathrm{IPL}}(N)$. Both are in the required form for the transformations in [2] and thus the leading term of $C_{\mathrm{IPL}}(N)$ can be deduced.

As a similar problem will arise in Section 4.2, the upper and lower bound, as well as the proofs, will be given for a more general recurrence equation:

$$
\begin{aligned}
&C_{\text{gen}}(1) = c_1,\ C_{\text{gen}}(2) = c_2,\\
&C_{\text{gen}}(3) = c_3,\ C_{\text{gen}}(4) = c_4 \qquad c_1,\ c_2,\ c_3,\ c_4 \in \mathbb{R}
\end{aligned}
$$

$$
\begin{aligned}
C_{\text{gen}}(N) = e(N-1) + d + \frac{6}{(N-2)(N-3)} \sum_{k=1}^{N-3} (N-k-2) C_{\text{gen}}(k),\\
\text{with } N > 4,\ e, d \in \mathbb{N}_{\geq 0}\\
\text{and } c_1 \leq c_2 \leq c_3 \leq c_4 \leq 4e + d + 2c_1 + c_2 \text{ (see monotony in the Appendix).}
\end{aligned}
\tag{4.3}
$$

4.1.1. Lower bound

Lemma 3. *If $\underline{C}$ is defined by:*

$$
\begin{aligned}
&\underline{C}_1 = 0,\\
&\underline{C}_N = e(N-2) + \frac{6}{N(N-1)} \sum_{k=1}^{N-2} (N-k-1)\underline{C}_k \qquad N \geq 2,
\end{aligned}
\tag{4.4}
$$

then it holds for $N \geq 3$ that

$$
\underline{C}_N = \frac{6}{5} e(N+1)H_{N+1} - \frac{66}{25} e(N+1) + \frac{3}{2} e. \tag{4.5}
$$

Proof. Equation (4.4) in [2] is the closed form of an identical recurrence equation. Using $pc_N = e(N-2)$, $a = e$, $b = -3e$ and $d = 0$ (4.3), the following closed form for $\underline{C}_N$ is obtained from (4.4):

$$
\begin{aligned}
\underline{C}_N &= a\frac{6}{5}(N+1)H_{N+1} + \left(-\frac{87}{50}a + \frac{3}{10}b + \frac{1}{10}d\right)(N+1) - \frac{1}{2}b\\
&= \frac{6}{5}e(N+1)H_{N+1} + \left(-\frac{87}{50}e - 3\frac{3}{10}e\right)(N+1) + 3\frac{1}{2}e\\
&= \frac{6}{5}e(N+1)H_{N+1} - \frac{66}{25}e(N+1) + \frac{3}{2}e
\end{aligned}
$$

□

Lemma 4. *If*

$$
\begin{aligned}
&c_1 \geq 0 = \underline{C}_1\\
&c_2 \geq 0 = \underline{C}_2\\
&c_3 \geq e = \underline{C}_3
\end{aligned}
$$

$c_4 \geq 2e = \underline{C}_4$,

then $C_{gen}(N) \geq \underline{C}_N$ *for all* $N \in \mathbb{N}_0$.

Proof. Proof by induction over N.

Induction Beginning:

For ($1 \leq N \leq 4$), the claim holds by preconditions of them.

Induction Hypothesis:

For an arbitrary fixed number N, holds for all $k \in \mathbb{N}$ with $k < N$ that $C_{\text{gen}}(k) \geq \underline{C}_k$.

Induction Step: for $N \geq 5$

$$\begin{aligned}
C_{\text{gen}}(N) &= e(N-1) + d + \frac{6}{(N-2)(N-3)} \sum_{k=1}^{N-3} (N-k-2) C_{\text{gen}}(k) \\
&= e(N-1) + d + \frac{(N-1)}{(N-3)} \cdot \frac{6}{(N-1)(N-2)} \sum_{k=1}^{N-3} (N-k-2) C_{\text{gen}}(k) \\
&\overset{\textbf{IV}}{\geq} e(N-1) + d + \frac{(N-1)}{(N-3)} \cdot \frac{6}{(N-1)(N-2)} \sum_{k=1}^{N-3} (N-k-2) \underline{C}_k \\
&= e(N-1) + d + \frac{(N-1)}{(N-3)} \Big(\underline{C}_{N-1} - e(N-3) \Big) \\
&= \frac{(N-1)}{(N-3)} \underline{C}_{N-1} + e(N-1) + d - \frac{(N-1)}{(N-3)} e(N-3) \\
&= \frac{(N-1)}{(N-3)} \underline{C}_{N-1} + d
\end{aligned}$$

Thus it needs to be shown that $\frac{(N-1)}{(N-3)} \underline{C}_{N-1} + d \geq \underline{C}_N$ for $d \in \mathbb{N}_{\geq 0}$.
Using the closed form of $\underline{C}_N$, which is valid for $N \geq 5$, I show that:

$$\frac{(N-1)}{(N-3)} (aN \cdot H_N + bN + c) + d \geq a(N+1) H_{N+1} + b(N+1) + c$$

(the constants are replaced by a, b and c for readability)

Proof by comparison of the different summands:

1. $\frac{(N-1)}{(N-3)} aN \cdot H_N \geq a(N+1) H_{N+1}$

$$
\begin{aligned}
&\iff (N-1)NH_N \geq (N-3)(N+1)H_{N+1} \\
&\iff (N^2-N)H_N \geq (N^2-2N-3)H_{N+1} \\
&\iff N^2(H_N - H_{N+1}) + N(-H_N + 2H_{N+1}) + 3H_{N+1} \geq 0 \\
&\iff -\frac{N^2}{(N+1)} + \frac{N}{(N+1)} + NH_{N+1} + 3H_{N+1} \\
&\qquad \geq -N(N+3)H_{N+1} + \frac{N}{(N+1)} \geq 0
\end{aligned}
$$

This holds as $(N+3)H_{N+1} \geq N$ for $N \geq 1$.

2. $\frac{(N-1)}{(N-3)} bN \geq b(N+1)$

$$
\begin{aligned}
&\iff (N-1)N \geq (N-3)(N+1) \\
&\iff (N^2-N) \geq (N^2-2N-3) \\
&\iff N+3 \geq 0
\end{aligned}
$$

This holds as $N \geq 4$.

3. $\frac{(N-1)}{(N-3)} c + d \geq c$

This holds as $\frac{(N-1)}{(N-3)} \geq 1$ for $N \geq 4$ and $d \in \mathbb{N}_{\geq 0}$.

Using 1, 2 and 3, it can be deduced that $\frac{(N-1)}{(N-3)}\underline{C}_{N-1} + d \geq \underline{C}_N$ and thus $C_{\text{gen}}(N) \geq \underline{C}_N$ for all $N \in \mathbb{N}_0$. □

4.1.2. Upper bound

Lemma 5. *Let $\overline{C}$ be defined by:*

$$
\begin{aligned}
&\overline{C}_0 = c_2,\ \overline{C}_1 = c_3,\ \overline{C}_2 = c_4 \\
&\overline{C}_N = e(N+3) + d + \frac{6}{N(N-1)} \sum_{k=0}^{N-2} (N-k-1)\overline{C}_k, \qquad N \geq 3.
\end{aligned}
\tag{4.6}
$$

For $n \geq 5$ holds that

$$
\begin{aligned}
\overline{C}_N = \frac{6}{5} e(N+1)H_{N+1} \left(\frac{3}{10}d - \frac{57}{50}e\right)(N+1) + \frac{N+1}{5}\left(\frac{3c_2 + 2c_3 + c_4}{2}\right) \\
- \frac{1}{2}(2e+d) + \frac{6}{\binom{n}{4}}\left(\frac{7}{30}e + \frac{1}{60}d - \frac{1}{6}\overline{C}_5 + \frac{1}{5}\overline{C}_4\right).
\end{aligned}
\tag{4.7}
$$

Proof. In [3] Appendix A, the closed form of an identical recurrence equation is deduced.

Using $T_N = e(N+3) + d$ i.e. ($a = e$, $b = (3e+d)$, $c_1 = c_2 = c_3 = 0$), $M = 2$, $\overline{C}_0^{\mathrm{IS}} = c_2$, $\overline{C}_1^{\mathrm{IS}} = c_3$ and $\overline{C}_2^{\mathrm{IS}} = c_4$ in (A.3), the following closed form of $\overline{C}_N$ is obtained for $N \geq M+3 = 5$:

$$\begin{aligned}
\overline{C}_N &= \frac{6}{5}a(N+1)H_{N+1} + \frac{N+1}{5}\left(\frac{19}{5}a + \frac{6(b-a)}{M+2} - 6aH_{M+2}\right) + \frac{a-b}{2} + \\
&\quad \frac{N+1}{5}\sum_{k=0}^{M}\frac{3M-2k}{\binom{M+2}{3}}\overline{C}_k^{\mathrm{IS}} + \frac{\binom{M+4}{5}}{\binom{N}{4}}\left(\frac{6}{5}a + \frac{2(a-b)}{M+3} + \frac{5b-17a}{2(M+4)}\right. \\
&\quad \left. - \frac{M-1}{M+4}\overline{C}_{M+3} + \frac{M-1}{M+3}\overline{C}_{M+2}\right) \\
&= \frac{6}{5}e(N+1)H_{N+1} + \frac{N+1}{5}\left(\frac{19}{5}e + \frac{3}{2}(2e+d) - 6eH_4\right) - \frac{1}{2}(2e+d) \\
&\quad + \frac{N+1}{5}\left(\frac{6}{\binom{4}{3}}\overline{C}_0^{\mathrm{IS}} + \frac{4}{\binom{4}{3}}\overline{C}_1^{\mathrm{IS}} + \frac{2}{\binom{4}{3}}\overline{C}_2^{\mathrm{IS}}\right) + \frac{\binom{6}{5}}{\binom{N}{4}}\left(\frac{6}{5}e - \frac{2}{5}(2e+d)\right. \\
&\quad \left. - \frac{1}{12}(2e-5d) - \frac{1}{6}\overline{C}_5 + \frac{1}{5}\overline{C}_4\right) \\
&= \frac{6}{5}e(N+1)H_{N+1}\left(\frac{3}{10}d - \frac{57}{50}e\right)(N+1) + \frac{N+1}{5}\left(\frac{3c_2 + 2c_3 + c_4}{2}\right) \\
&\quad - \frac{1}{2}(2e+d) + \frac{6}{\binom{n}{4}}\left(\frac{7}{30}e + \frac{1}{60}d - \frac{1}{6}\overline{C}_5 + \frac{1}{5}\overline{C}_4\right).
\end{aligned}$$

□

Lemma 6. *$\overline{C}_n \geq C_{gen}(n+2)$ holds for all $n \in \mathbb{N}$.*

Proof. The proof is conducted by induction over N.

Induction Beginning:

$N = 0: \;\; \overline{C}_0 = c_2 \geq c_2 = C_{\mathrm{gen}}(2).$

$N = 1: \;\; \overline{C}_1 = c_3 \geq c_3 = C_{\mathrm{gen}}(3).$

$N = 2: \;\; \overline{C}_2 = c_4 \geq c_4 = C_{\mathrm{gen}}(4).$

Induction Hypothesis:

For an arbitrary fixed number n, holds for all $k \in \mathbb{N}$ with $k \leq N-2$ that $\overline{C}_k \geq C_{\mathrm{gen}}(k+2)$.

Induction Step:

$$\begin{aligned}
C_{\text{gen}}(N+2) &= e(N+3) + d + \frac{6}{N(N-1)} \sum_{k=1}^{N-1} (N-k) C_{\text{gen}}(k) \\
&= e(N+3) + d + \frac{6}{N(N-1)} \sum_{k=0}^{N-2} (N-k-1) C_{\text{gen}}(k+1) \\
&\overset{(*)}{\leq} e(N+3) + d + \frac{6}{N(N-1)} \sum_{k=0}^{N-2} (N-k-1) C_{\text{gen}}(k+2) \\
&\overset{\textbf{IV}}{\leq} e(N+3) + d + \frac{6}{N(N-1)} \sum_{k=0}^{N-2} (N-k-1) \overline{C}_k = \overline{C}_N.
\end{aligned}$$

(*): The monotony of $C_{\text{gen}}(N)$ is proven in the Appendix. □

4.1.3. Approximation of Expected Internal Path Length

Using the results of Subsections 4.1.1 and 4.1.2, the closed form of $C_{\text{IPL}}(N)$ can sandwiched between the closed forms of a lower bound and an upper bound.

Lemma 7. *$\underline{C}_{IPL}(N) \leq C_{IPL}(N) \leq \overline{C}_{IPL}(N)$ for $N \geq 7$,*
if $\underline{C}_{IPL}(N) = \frac{6}{5}(N+1)H_{N+1} - \frac{66}{25}(N+1) + \frac{3}{2}$ and
$\overline{C}_{IPL}(N) = \frac{6}{5}(N+1)H_{N+1} - \frac{7}{50}(N+1) - 1$.

Proof. Both bounds follow from Lemmas 3 and 4 (lower bound) resp. 5 and 6 (upper bound) by setting $e = 1$, $d = 0$ and

$$\begin{aligned}
&c_1 = 0 \geq 0,\ c_2 = 1 \geq 0, \\
&c_3 = 2 \geq e = 1,\ c_4 = 3 \geq 2e = 2 \\
&(\text{with } c_1 \leq c_2 \leq c_3 \leq c_4 \leq 2 = 2e + d + 2c_1 + c_2)
\end{aligned}$$

in the definition of $C_{\text{gen}}(N)$.
The lower bound follows directly and for the upper bound, the following calculation

can be made using that $\overline{C}_{\text{IPL}}(4) = 12$ and $\overline{C}_{\text{IPL}}(5) = \frac{79}{5}$:

$$\begin{aligned}
\overline{C}_{\text{IPL}}(N) &= \frac{6}{5}e(N+1)H_{N+1}\left(\frac{3}{10}d - \frac{57}{50}e\right)(n+1) + \frac{N+1}{5}\left(\frac{3c_2+2c_3+c_4}{2}\right) \\
&\quad - \frac{1}{2}(2e+d) + \frac{6}{\binom{n}{4}}\left(\frac{7}{30}e + \frac{1}{60}d - \frac{1}{6}\overline{C}_5 + \frac{1}{5}\overline{C}_4\right) \\
&= \frac{6}{5}(n+1)H_{N+1} - \frac{57}{50}(N+1) + \frac{N+1}{5}\left(\frac{3+4+3}{2}\right) - 1 \\
&\quad + \frac{6}{\binom{N}{4}}\underbrace{\left(\frac{7}{30} - \frac{1}{6}\frac{79}{5} + \frac{1}{5}12\right)}_{=0} \\
&= \frac{6}{5}(N+1)H_{N+1} - \frac{7}{50}(N+1) - 1.
\end{aligned}$$

□

Using the closed forms of $\overline{C}_{\text{IPL}}(N-2)$ and $\underline{C}_{\text{IPL}}(N)$, an approximate closed form of $C_{\text{IPL}}(N)$ can be deduced. As $\overline{C}_{\text{IPL}}(N-2)$ and $\underline{C}_{\text{IPL}}(N)$ do not have the same leading term, I will first transform $\overline{C}_{\text{IPL}}(N-2)$ to allow a better comparison.

$$\begin{aligned}
\overline{C}_{\text{IPL}}(N-2) &= \frac{6}{5}(N-1)H_{N-1} - \frac{7}{50}(N-1) - 1 \\
&= \frac{6}{5}(N+1)H_{N-1} - 2\frac{6}{5}H_{N-1} - \frac{7}{50}(N+1) + 2\frac{7}{50} - 1 \\
&= \frac{6}{5}(N+1)\left(H_{N+1} - \frac{1}{N+1} - \frac{1}{N}\right) - \frac{7}{50}(N+1) - \frac{12}{5}H_{N-1} - \frac{18}{25} \\
&= \frac{6}{5}(N+1)H_{N+1} - \frac{6}{5}\frac{(N+1)}{(N+1)} - \frac{6}{5}\frac{(N+1)}{N} - \frac{7}{50}(N+1) - \frac{12}{5}H_{N-1} - \frac{18}{25} \\
&= \frac{6}{5}(N+1)H_{N+1} - \frac{6}{5} - \frac{6}{5} - \frac{6}{5}\frac{1}{N} - \frac{7}{50}(N+1) - \frac{12}{5}H_{N-1} - \frac{18}{25} \\
&= \frac{6}{5}(N+1)H_{N+1} - \frac{7}{50}(N+1) - \frac{12}{5}H_{N-1} - \frac{6}{5N} - \frac{78}{25}.
\end{aligned}$$

Using this representation of $\overline{C}_{\text{IPL}}(N-2)$ and $\underline{C}_{\text{IPL}}(N)$, the following approximate closed form of $C_{\text{IPL}}(N)$ can be given:

Theorem 5. *$C_{IPL}(N) = \frac{6}{5}N \cdot H_N + a(N+1) + \mathcal{O}\big(\log(N)\big)$ with $-\frac{66}{25} \leq a \leq -\frac{7}{50}$.*

Proof. With the closed forms of $\overline{C}_{\text{IPL}}(N-2)$ rsp. $\underline{C}_{\text{IPL}}(N)$ and $\frac{12}{5}H_{N-1} + \frac{6}{5N} + \frac{78}{25}, \frac{3}{2} \in \mathcal{O}\big(\log(N)\big)$ (eq. (6.66) in [5]), the following calculation can be made:

$$\begin{aligned}
C_{\mathrm{IPL}}(N) &= \frac{6}{5}(N+1)H_{N+1} + a(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{6}{5}N \cdot H_{N+1} + \frac{6}{5}H_{N+1} + a(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{6}{5}N\left(H_N + \frac{1}{N+1}\right) + \frac{6}{5}H_{N+1} + a(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{6}{5}N \cdot H_N + \frac{6}{5}\frac{N}{N+1} + \frac{6}{5}H_{N+1} + a(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{6}{5}N \cdot H_N + a(N+1) + \mathcal{O}\big(\log(N)\big), \qquad N \geq 7.
\end{aligned}$$

□

For a better sense of order of magnitude, the asymptotic representation is given:

Theorem 6. *The asymptotic representation of $C_{IPL}(N)$ is given by:*

$$C_{IPL}(N) = \frac{6}{5}N\ln(N) + \left(\frac{6\gamma}{5} + a\right)N + \mathcal{O}\big(\log(N)\big) \qquad N \to \infty$$

with $-\frac{66}{25} \leq a \leq -\frac{7}{50}$ and γ the Euler-Mascheroni constant.

Proof. Using the asymptotic estimate: (eq. (6.66) in [5])

$$H_n = \ln(n) + \gamma + \mathcal{O}\left(\frac{1}{n}\right)$$

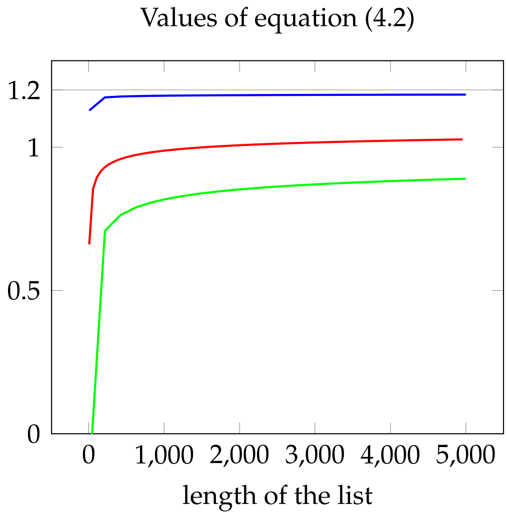
the following calculation can be made for $N \to \infty$:

$$\begin{aligned}
C_{\mathrm{IPL}}(N) &= \frac{6}{5}N \cdot H_N + a(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{6}{5}N(\ln(N) + \gamma + \mathcal{O}(\frac{1}{N})) + a(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{6}{5}N\ln(N) + \frac{6\gamma}{5}N + \mathcal{O}(1) + aN + a + \mathcal{O}\big(\log(N)\big) \\
&= \frac{6}{5}N\ln(N) + (\frac{6\gamma}{5} + a)N + \mathcal{O}\big(\log(N)\big).
\end{aligned}$$

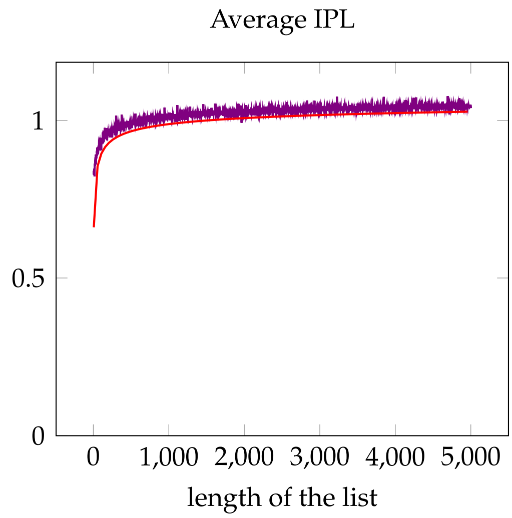
□

Figure 4.1(a) contains the values of equation (4.2) (red curve) for $1 \leq N \leq 1000$, normalized by $N\ln(N)$. Unfortunately, the convergence to the prefactor $\frac{6}{5} = 1.2$ (gray line) is very slow, which explains why the values of (4.2) are that far away from $\frac{6}{5}$ for "small" N. The blue resp. green curve represent the upper resp. lower bound. Even for $N = 1000$, they still lie very far apart, which does not allow a precise asymptotic.

Figure 4.1(b) displays the empiric data, collected by generating 100 2-jumplists for the sizes 1 to 1000 resp., taking the average IPL of each sample and normalizing by $N \ln(N)$. Unfortunately, the convergence is also very slow for these values.



(a) Values of equation (4.2) (red), $\overline{C}_{\text{IPL}}(N-2)$ (blue) and $\underline{C}_{\text{IPL}}(N)$ (green) for $1 \leq N \leq 1000$, normalized by $N \ln(N)$.



(b) Average IPL of 100 2-jumplists of sizes 2-1000, normalized by $n \ln(n)$ (violet) compared to the normalized values of equation (4.2).

Figure 4.1.: Plots for IPL.

4.2. Expected Number of Comparisons

As mentioned above, the expected number of comparisons needed for a successful search of all the elements in a 2-jumplist of length N (header included) is considered to avoid any dependence on the position of an element. As for the IPL, the expected number of comparisons will be given as a recurrence equation:

$$C_{\text{comp\_all}}(1) = 4,\ C_{\text{comp\_all}}(2) = 9,\ C_{\text{comp\_all}}(3) = 15,\ C_{\text{comp\_all}}(4) = 22$$

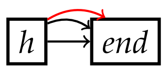
$$\begin{aligned} C_{\text{comp\_all}}(n) = 4 &+ \frac{1}{\binom{n-2}{2}} \sum_{i=1}^{n-3} \sum_{j=1}^{n-2-i} (3i + 2j + 1(n-1-i-j)) \\ &+ \frac{1}{\binom{n-2}{2}} \sum_{i=1}^{n-3} \sum_{j=1}^{n-2-i} (C_{\text{comp\_all}}(i) + C_{\text{comp\_all}}(j) + C_{\text{comp\_all}}(n-1-i-j))), \\ &\hfill (n > 4). \end{aligned} \tag{4.8}$$

To access the nodes in the jump2-, jump- or next-list resp., one, two or three comparison are needed resp. (see **Search** lines 3, 6 and 11). To access the current header, even 4 comparisons are needed (see **Search** lines 3, 6 , 11 and 16).

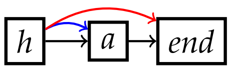
For all the nodes in the jump2-, jump- and next-list, the costs to use the jump2-, jump- and next-pointer, resp., need to be added. Therefore, the toll function consists of the summand 4 to access the header and the double sum, which traverses all the possible values for the *nsize*, *jsize* and *j2size*. The factor $1/\binom{N-2}{2}$ before this double sum averages over the possible pointer targets of the header.

For the initial conditions, refer to Figure 4.2. As *list.header* contains no label, it will not

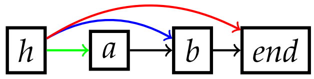
(a) Initial condition for $N = 1$. 4 comparison are needed to access h.



(b) Initial condition for $N = 2$. 1 comparison is needed to follow the red arrow and 4 resp. to access *end* and h.



(c) Initial condition for $N = 3$. For *end*, 5 comparisons are needed in total (See (a)). For a, 2 comparison are needed to follow the blue arrow, 4 to access the node and 4 for h.



(d) Initial condition for $N = 4$. For *end* resp. b, 5 resp. 6 comparisons are needed in total (See (a) and (b)). For a, 3 comparison are needed to follow the green arrow, 4 to access the node and 4 for h.

Figure 4.2.: Explanation of the initial conditions of equation (4.8).

be accessed during the search. Therefore, the four comparisons to access *list.header* need to be subtracted from $(C_{\text{comp\_all}}(N))$, resulting in $C_{\text{comp\_all}}(N) - 4$ for the expected number of comparisons.

By using the same simplifications for second the double sum as in equation (4.2) and the following calculations, a simplified form of $C_{\text{comp\_all}}(N)$ can be deduced:

$$
\begin{aligned}
&4+\frac{1}{\binom{N-2}{2}}\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}(3i+2j+1(N-1-i-j))\\
&=4+\frac{3}{\binom{N-2}{2}}\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}i+\frac{2}{\binom{N-2}{2}}\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}j+\frac{1}{\binom{N-2}{2}}\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}(N-1-i-j)\\
&=4+\frac{3}{\binom{N-2}{2}}\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}i+\frac{2}{\binom{N-2}{2}}\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}j+\frac{1}{\binom{N-2}{2}}\sum_{i=1}^{N-3}\sum_{j=1}^{N-2-i}j\\
&=4+\frac{6}{(N-2)(N-3)}\sum_{i=1}^{N-3}i\sum_{j=1}^{N-2-i}1+\frac{4}{(N-2)(N-3)}\sum_{j=1}^{N-3}j\sum_{i=1}^{N-2-j}1+\\
&\quad\frac{2}{(N-2)(N-3)}\sum_{j=1}^{N-3}j\sum_{i=1}^{N-2-j}1\\
&=4+\frac{6}{(N-2)(N-3)}\frac{1}{6}(N-1)(N-2)(N-3)+\frac{4}{(N-2)(N-3)}\\
&\quad\cdot\frac{1}{6}(N-1)(N-2)(N-3)+\frac{2}{(N-2)(N-3)}\frac{1}{6}(N-1)(N-2)(N-3)\\
&=4+(N-1)+\frac{2(N-1)}{3}+\frac{(N-1)}{3}\\
&=2(N-1)+4=2(N+1).
\end{aligned}
$$

Intuitively, this result can be explained as follows: Every node is with the same probability in either the next-, jump- or jump2-list. Therefore, the expected number of comparisons needed to follow either the next-, jump- or jump2-pointer of the header is 2 for each of the $(N-1)$ nodes. Adding the 4 comparisons which are needed to access the header, finally results in $2(N+1)$.

$$
\begin{aligned}
&C_{\text{comp\_all}}(1)=4,\ C_{\text{comp\_all}}(2)=9,\ C_{\text{comp\_all}}(3)=15,\ C_{\text{comp\_all}}(4)=22,\\
&C_{\text{comp\_all}}(N)=2(N+1)+\frac{6}{(N-2)(N-3)}\sum_{k=1}^{N-3}(N-k-2)C_{\text{comp\_all}}(k),\qquad N\geq 5.
\end{aligned}
\tag{4.9}
$$

4.2.1. Approximation of Expected Number of Comparisons

To estimate the closed form of $C_{\text{comp\_all}}(N)$, the same method as in 4.1.3 is used: determine an upper and lower bound for $C_{\text{comp\_all}}(N)$ and use their closed form for the estimation.

Lemma 8. *$\underline{C}_{comp\_all}(N) \le C_{comp\_all}(N) \le \overline{C}_{comp\_all}(N)$ for $N \ge 7$, if $\underline{C}_{comp\_all}(N) = \frac{12}{5}(N+1)H_{N+1} - \frac{132}{25}(N+1) + 3$ and $\overline{C}_{comp\_all}(N) = \frac{12}{5}(N+1)H_{N+1} + \frac{341}{50}(N+1) - 4$.*

Proof. The lemma follows from Lemmas 3 and 4 (lower bound) resp. Lemmata 5 and 6 (upper bound) by setting $e = 2$, $d = 4$ and

$$\begin{aligned}
&c_1 = 4 \ge 0,\ c_2 = 9 \ge 0,\\
&c_3 = 15 \ge e = 2,\ c_4 = 22 \ge 2e = 4\\
&(\text{with } c_1 \le c_2 \le c_3 \le c_4 \le 29 = 2e + d + 2c_1 + c_2)
\end{aligned}$$

in the definition of $C_{\text{gen}}(N)$. The lower bound follows directly and for the upper bound, the following calculation can be made using $\overline{C}_{\text{comp\_all}}(4) = \frac{115}{2}$ and $\overline{C}_{\text{comp\_all}}(5) = \frac{361}{5}$:

$$\begin{aligned}
\overline{C}_{\text{comp_all}}(N) &= \frac{6}{5}e(N+1)H_{N+1} + \left(\frac{3}{10}d - \frac{57}{50}e\right)(N+1) + \frac{N+1}{5}\left(\frac{3c_2 + 2c_3 + c_4}{2}\right)\\
&\quad - \frac{1}{2}(2e+d) + \frac{6}{\binom{n}{4}}\left(\frac{7}{30}e + \frac{1}{60}d - \frac{1}{6}\overline{C}_5 + \frac{1}{5}\overline{C}_4\right)\\
&= \frac{12}{5}(N+1)H_{N+1} + \left(\frac{6}{5} - \frac{57}{25}\right)(N+1) + \frac{N+1}{5}\left(\frac{27+30+22}{2}\right)\\
&\quad - 4 + \frac{6}{\binom{N}{4}}\underbrace{\left(\frac{7}{15} + \frac{1}{15} - \frac{1}{6}\frac{361}{5} + \frac{1}{5}\frac{115}{2}\right)}_{=0}\\
&= \frac{12}{5}(N+1)H_{N+1} + \frac{341}{50}(N+1) - 4.
\end{aligned}$$

□

Again, this representation of $\overline{C}_{\text{comp\_all}}(N-2)$ is not very appealing, because it has not the same leading term as $\underline{C}_{\text{comp\_all}}(N)$. Therefore, $\overline{C}_{\text{comp\_all}}(N-2)$ will be modified first:

$$\begin{aligned}
\overline{C}_{\text{comp_all}}(N-2) &= \frac{12}{5}(N-1)H_{N-1} + \frac{341}{50}(N-1) - 4 \\
&= \frac{12}{5}(N+1)H_{N-1} - 2\frac{12}{5}H_{N-1} + \frac{341}{50}(N+1) - 2\frac{341}{50} - 4 \\
&= \frac{12}{5}(N+1)\left(H_{N+1} - \frac{1}{N+1} - \frac{1}{N}\right) + \frac{341}{50}(N+1) - \frac{24}{5}H_{N-1} \\
&\quad - \frac{441}{25} \\
&= \frac{12}{5}(N+1)H_{N+1} - \frac{12}{5}\frac{(N+1)}{(N+1)} - \frac{12}{5}\frac{(N+1)}{N} + \frac{341}{50}(N+1) \\
&\quad - \frac{24}{5}H_{N-1} - \frac{441}{25} \\
&= \frac{12}{5}(N+1)H_{N+1} - \frac{12}{5} - \frac{12}{5} - \frac{12}{5}\frac{1}{N} + \frac{341}{50}(N+1) \\
&\quad - \frac{24}{5}H_{N-1} - \frac{441}{25} \\
&= \frac{12}{5}(N+1)H_{N+1} + \frac{341}{50}(N+1) - \frac{24}{5}H_{N-1} - \frac{12}{5N} - \frac{561}{25}.
\end{aligned}$$

Using this representation of $\overline{C}_{\text{comp\_all}}(N-2)$ and $\underline{C}_{\text{comp\_all}}(N)$, the following approximate closed form of $C_{\text{comp\_all}}(N)$ can be given:

Theorem 7. *$C_{comp\_all}(n) = \frac{12}{5}nH_n + a'(n+1) + \mathcal{O}(\log(n))$ with $-\frac{132}{25} \leq a' \leq \frac{341}{50}$.*

Proof. With the closed forms of $\overline{C}_{\text{comp\_all}}(n-2)$ rsp. $\underline{C}_{\text{comp\_all}}(n)$ and $\frac{24}{5}H_{n-1} + \frac{12}{5n} + \frac{441}{25}, 4 \in \mathcal{O}\big(\log(N)\big)$(eq. (6.66) in [5]), the following calculation can be made:

$$\begin{aligned}
C_{\text{comp_all}}(N) &= \frac{12}{5}(N+1)H_{N+1} + a'(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{12}{5}N \cdot H_{N+1} + \frac{12}{5}H_{N+1} + a'(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{12}{5}N\left(H_N + \frac{1}{N+1}\right) + \frac{12}{5}H_{N+1} + a'(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{12}{5}N \cdot H_N + \frac{12}{5}\frac{N}{N+1} + \frac{12}{5}H_{N+1} + a'(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{12}{5}N \cdot H_N + a'(N+1) + \mathcal{O}\big(\log(N)\big).
\end{aligned}$$

□

For a better sense of order of magnitude, the asymptotic representation is given in Theorem 8.

Theorem 8. *The asymptotic representation of $C_{comp\_all}(N)$ is given by:*

$$C_{comp\_all}(N) = \frac{12}{5}N\ln(N) + \left(\frac{12\gamma}{5} + a'\right)N + \mathcal{O}\big(\log(N)\big) \qquad N \to \infty$$

with $-\frac{132}{25} \leq a' \leq \frac{341}{50}$ and γ the Euler-Mascheroni constant.

Proof. Using the asymptotic estimate: (eq. (6.66) in [5])

$$H_n = \ln(n) + \gamma + \mathcal{O}(\frac{1}{N})$$

the following calculation can be made:

$$\begin{aligned}
C_{\text{comp_all}}(N) &= \frac{12}{N}N \cdot H_N + a'(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{12}{5}N\Big(\ln(N) + \gamma + \mathcal{O}(\frac{1}{N})\Big) + a'(N+1) + \mathcal{O}\big(\log(N)\big) \\
&= \frac{12}{5}N\ln(N) + \frac{12\gamma}{5}N + \mathcal{O}(1) + a'N + a' + \mathcal{O}\big(\log(N)\big) \\
&= \frac{12}{5}N\ln(N) + (\frac{12\gamma}{5} + a')N + \mathcal{O}\big(\log(N)\big), \qquad N \to \infty.
\end{aligned}$$

□

In order to estimate the number of comparisons for a search of a single element, $C_{\text{comp\_all}}(N) - 4$ needs to be divided by the length of the list (without the header).

As mentioned in [1], the number of comparisons dominate the runtime of the search, leading to the following theorem:

Theorem 9.

1. *The expected number of comparisons for the search of an element in a 2-jumplist of length N is given by $C_{comp}(N) = \frac{12}{5}\ln(N+1) + (\frac{12}{5}\gamma + a') + \mathcal{O}(\frac{\log(N)}{N})$ with $-\frac{132}{25} \leq a' \leq \frac{341}{50}$.*
2. *The search in a 2-jumplist has an expected time complexity of $\mathcal{O}(\log(N))$.*

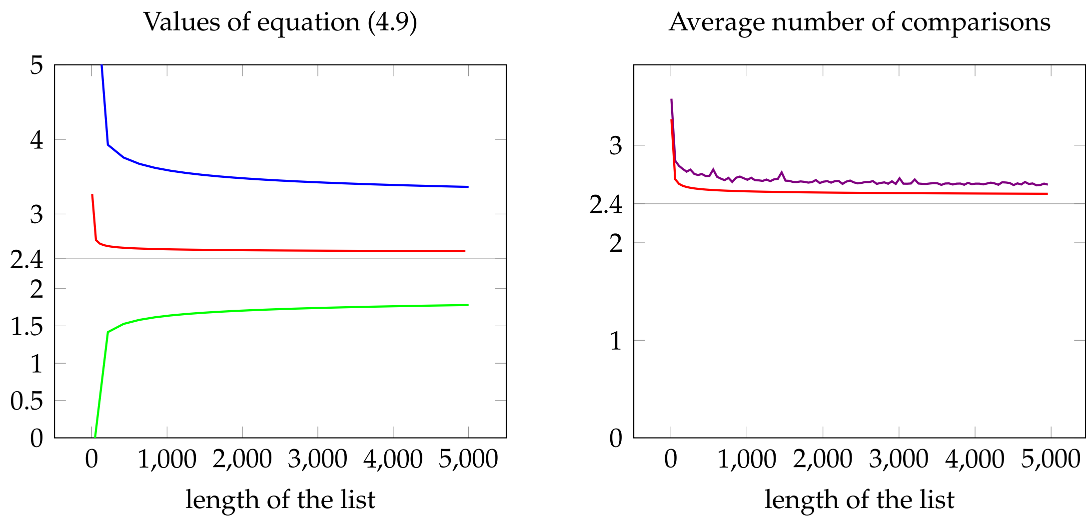
Proof.

$$
\begin{aligned}
C_{\text{comp}}(N) &= \frac{C_{\text{comp\_all}}(N) - 4}{N} \\
&= \frac{12}{5}\frac{N\ln(N)}{N} + \left(\frac{12}{5}\gamma + a'\right)\frac{N}{N} + \mathcal{O}\left(\frac{\log(N)}{N}\right) - \frac{4}{N} \\
&= \frac{12}{5}\ln(N) + \left(\frac{12}{5}\gamma + a'\right) + \mathcal{O}\left(\frac{\log(N)}{N}\right) - \frac{4}{N} \\
&= \frac{12}{5}\ln(N) + \left(\frac{12}{5}\gamma + a'\right) + \mathcal{O}\left(\frac{\log(N)}{N}\right) \in \mathcal{O}(\log(N)) \qquad N \to \infty
\end{aligned}
$$

□

Figure 4.3(a) contains the values of equation (4.9) (red curve) for $1 \leq N \leq 1000$, normalized by $N\ln(N)$. As for equation (4.2), the convergence to the prefactor $\frac{12}{5}$ is very slow and the upper bound (blue) is very far apart from the lower bound (green).

Figure 4.1(b) displays empirical data, collected by generating 100 2-jumplists for the sizes 1 to 1000 resp., averaging the number of comparisons needed for a search for every key over the sample and normalizing by $N\ln(N)$. Unfortunately, the convergence is also very slow for these values.



(a) Values of equation (4.9) (red), $\overline{C}_{\text{comp\_all}}(N-2)$ (blue) and $\underline{C}_{\text{comp\_all}}(N)$ (green) for $7 \leq N \leq 1000$, normalized by $N\ln(N)$.

(b) Average number of comparisons of 100 2-jumplists of sizes 2-1000, normalized by $N\ln(N)$ compared to the normalized values of equation (4.9).

Figure 4.3.: Plots for the number of comparisons.

5. Complexity Analysis

5.1. Generation

The major work for the generation of a 2-jumplist is provided by **rebalance**. The **Preprocessing** needs a linear number of operations, because a singly-linked list needs to be generated.
To determine the runtime of **rebalance** in $\mathcal{O}$-term notation, the costs of a call of **rebalance** on a node h with a sublist of length n will be given as a recurrence equation.

$$\begin{aligned}
C_{\text{reb}}(1) &= r_1,\ C_{\text{reb}}(2) = r_2,\ C_{\text{reb}}(3) = r_3, \qquad r_1, r_2, r_3 \in \mathbb{R}^+ \\
C_{\text{reb}}(n) &= r + \frac{1}{\binom{n-2}{2}} \sum_{i=1}^{n-3} \sum_{j=1}^{n-2-i} \Big(C_{\text{reb}}(i) + C_{\text{reb}}(j) + C_{\text{reb}}(n-1-i-j) \Big) \\
&= r + \frac{6}{(n-2)(n-3)} \sum_{k=1}^{n-3} (n-k-2) C_{\text{reb}}(k), \qquad r \in \mathbb{R}^+.
\end{aligned} \tag{5.1}$$

For every node in the list, a constant number of operations (mainly choosing the target of the pointers and assigning them) is needed. Therefore, the toll function consists of a constant r. In the double sum, the recursive calls (lines 15-17) for all the possible values of *nsize*, *jsize* and *j2size* are handled. The prefactor $1/(\binom{n-2}{2})$ of the sum averages over all the possible pointer configurations.
To prove that the expected runtime of **rebalance** is in $\mathcal{O}(n)$, I show that $C_{\text{reb}}(n) \leq an$ for $n \in \mathbb{N}_{\geq 1}$ and a suitable constant a:

Lemma 9. *Let* $a = \max\{r_1, \frac{r_2}{2}, \frac{r_3}{3}, r\}$ *for* $n \in \mathbb{N}_{\geq 1}$. *Then* $C_{reb}(n) \leq an$ *for* $n \in \mathbb{N}_{\geq 1}$.

Proof. The proof is conducted by induction.

Induction Beginning:

$n = 1: \ C_{\text{usu}}(1) = r_1 \leq a$
$n = 2: \ C_{\text{usu}}(2) = r_2 \leq 2a$
$n = 3: \ C_{\text{usu}}(3) = r_3 \leq 3a$

Induction Hypothesis:

For an arbitrary fixed number n, holds for all $k \in \mathbb{N}$ with $k < n$ that $C_{\text{usu}}(k) \leq ak$.

Induction Step:

$$\begin{aligned}
C_{\text{reb}}(n) =& r + \frac{6}{(n-2)(n-3)} \sum_{k=1}^{n-3} (n-k-2) C_{\text{reb}}(k) \\
&\overset{\textbf{IV}}{\leq} r + \frac{6}{(n-2)(n-3)} \sum_{k=1}^{n-3} (n-k-2) ak \\
&= r + \frac{6a}{(n-2)(n-3)} \cdot \frac{(n-1)(n-2)(n-3)}{6} \\
&= r + a(n-1) \leq an.
\end{aligned}$$

□

Corollary 2. *The generation of a 2-jumplist of length $(n+1)$ has an expected runtime of $\mathcal{O}(n)$.*

5.2. Insertion

The time complexity of insertion is dominated by three sub-procedures: the (unsuccessful) search, restore_unif/bend_procedure and usurper. The rest of the insertion, the preprocessing, is done in constant time.

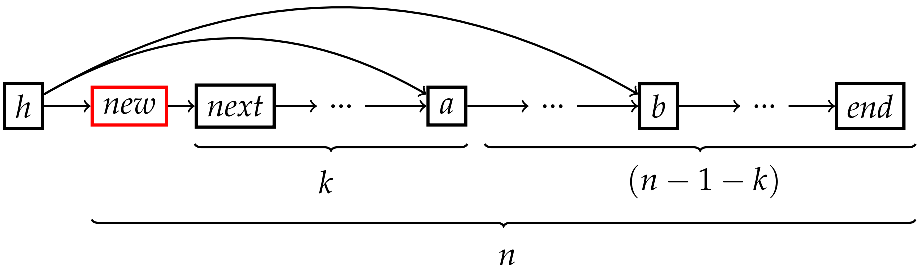
The complexity of the search, $\mathcal{O}(\log n)$, has already been determined in Theorem 9. The complexity of the other two dominating procedures will be determined in the following, leading to Theorem 10.

Theorem 10. *The insertion of a node in a 2-jumplist of length n has expected time complexity of $\mathcal{O}(\log(n))$*

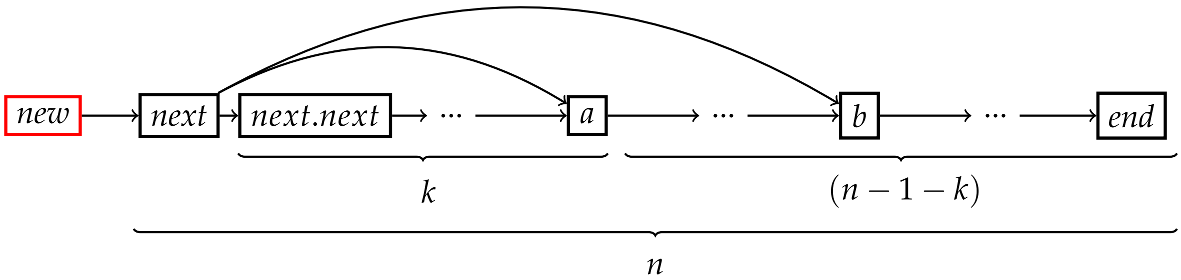
Proof. See Chapter 4 and Sections 5.2.1 and 5.2.2. □

5.2.1. Usurper

The cost of a call of **usurp** on a header with a sublist of length n (see Figure 5.1 for further illustrations), can be expressed by a recurrence equation. Although usurper contains two phases, they are similar enough to be expressed by the same equation.



(a) First phase



(b) Second phase. *new* represents the new node or a node whose pointers have been stolen in the previous iteration.

Figure 5.1.: Illustration of the sizes used in equation (5.2).

$$
\begin{aligned}
&C_{\text{usu}}(2) = k_2,\ C_{\text{usu}}(3) = k_3,\\
&C_{\text{usu}}(4) = \frac{2}{4-1}C_{\text{reb}}(4) + (1 - \frac{2}{4-1})C_{\text{usu}}(2) = k_4 \qquad k_2, k_3, k_4 \in \mathbb{R}^+
\end{aligned}
$$

$$
\begin{aligned}
C_{\text{usu}}(n) &= \overbrace{\frac{2}{(n-1)}C_{\text{reb}}(n)}^{\leq l} + (1 - \frac{2}{(n-1)}) \sum_{k=2}^{n-2} \frac{(n-1-k)}{\binom{n-2}{2}} C_{\text{usu}}(k) \qquad (5.2)\\
&\leq l + \frac{(n-3)}{(n-1)} \frac{2}{(n-2)(n-3)} \sum_{k=2}^{n-2} (n-1-k) C_{\text{usu}}(k)\\
&= l + \frac{1}{\binom{n-1}{2}} \sum_{k=2}^{n-2} (n-1-k) C_{\text{usu}}(k). \qquad (\text{n} \geq 5),\ l \in \mathbb{R}^+
\end{aligned}
$$

In every recursion of usurper, the algorithm chooses whether or not to bend the pointers of h resp. *new*.

The bending occurs with probability $\frac{2}{n-1}$ and implies a call of **rebalance** on parts of the sublist of h resp. *new*. As **rebalance** has a complexity of $\mathcal{O}(n)$, there exists an

$l \in \mathbb{R}^+$ s. t. $C_{\text{reb}} \leq l \cdot n$. This argumentation explains the first addend of the recursion.

If however the algorithm chooses not to bend any pointer, which it does with probability $(1 - \frac{2}{n-1})$, a recursive call is executed on node *next* and its sublist. The length k of this sublist, which determines the complexity of this call, depends on the *nsize* of the previous node an thus on his jump-pointer.
If the algorithm is in the first phase, the size k of the sublist of the next recursive call is determined by the jump-pointer of h. Before the insertion of *new*, h had a sublist of length $(n-1)$, thus having $\binom{n-2}{2}$ possibilities for the choice of its pointers (assume that $(n-1) \geq 4$, the special cases will be handled in the initial conditions). To ensure that the next recursive call is on a sublist of length k, one pointer of h needs to point to the k-th node in its sublist (before the insertion) and the other one needs to point to a node located after, having $(n-1-k)$ possible targets, leading to the factor $(n-1-k)/\binom{n-2}{2}$ in the sum.
The sum starts at 2 as a sublist of length one cannot occur in the first phase (the sublist contains at least *new* and *next*) and during the second phase, a sublist of length one can only occur if during the previous recursion, the sublist had length two or three, which are handled by the initial conditions.
The initial conditions for $(n = 2)$ resp. $(n = 3)$ represent the special cases (a) resp. (b) in Figure 3.5. The one for $(n = 4)$ ensures that $(n-1) \geq 4$ to avoid special cases in the sum.

To prove that the complexity of **usurp** is in $\mathcal{O}(\text{ld}(n))$, I show that
$C_{\text{usu}}(n) \leq b\,\text{ld}(n)$ for $n \in \mathbb{N}_{\geq 2}$ and a suitably chosen constant b.

Lemma 10. *Let* $b = \max\{\frac{k_2}{\text{ld}(2)}, \frac{k_3}{\text{ld}(3)}, \frac{k_4}{\text{ld}(4)}, \frac{4}{3}l\}$. *Then* $C_{usu}(n) \leq b\,\text{ld}(n)$ *for* $n \in \mathbb{N}_{\geq 2}$.

Proof. The proof is conducted by induction.

Induction Beginning:

$$\begin{aligned}
n = 2: &\quad C_{\text{usu}}(2) = k_2 \leq b\,\text{ld}(2)\\
n = 3: &\quad C_{\text{usu}}(3) = k_3 \leq b\,\text{ld}(3)\\
n = 4: &\quad C_{\text{usu}}(4) = k_4 \leq b\,\text{ld}(4)
\end{aligned}$$

Induction Hypothesis:

For an arbitrary fixed number n, holds for all $k \in \mathbb{N}$ with $k < n$ that $C_{\text{usu}}(k) \leq b\,\text{ld}(k)$.

Induction Step:

For $s \in \mathbb{N}_{\geq 1}$ holds that $\text{ld}(s) - \text{ld}(\frac{s}{2}) = 1$.
Therefore, it holds for $(1 \leq k \leq \frac{n}{2})$ that $\text{ld}(n) - \text{ld}(k) \geq 1$ and for $(k \geq \frac{n}{2})$ that $\text{ld}(n) - \text{ld}(k) \leq 1$.

$$
\begin{aligned}
C_{\text{usu}}(n) &\leq l + \frac{1}{\binom{n-1}{2}} \sum_{k=2}^{n-2} (n-1-k) C_{\text{usu}}(k) \\
&\overset{\textbf{IV}}{\leq} l + \frac{2}{(n-1)(n-2)} \sum_{k=2}^{n-2} (n-1-k)(b\,\mathrm{ld}(k)) \\
&= l + \frac{2}{(n-1)(n-2)} \sum_{k=1}^{n-2} (n-1-k) b(\mathrm{ld}(n) - (\mathrm{ld}(n) - \mathrm{ld}(k))) \\
&= l + \frac{2b\,\mathrm{ld}(n)}{(n-1)(n-2)} \sum_{k=1}^{n-2} (n-1-k) \\
&\qquad - \frac{2b}{(n-1)(n-2)} \sum_{k=1}^{n-2} (n-1-k)(\mathrm{ld}(n) - \mathrm{ld}(k))) \\
&\leq l + \frac{2b\,\mathrm{ld}(n)}{(n-1)(n-2)} \sum_{k=1}^{n-2} (n-1-k) - \frac{2b}{(n-1)(n-2)} \sum_{k=1}^{n/2} (n-1-k)1 \\
&= l + \frac{2b\,\mathrm{ld}(n)}{(n-1)(n-2)} \frac{1}{2}(n-1)(n-2) - \frac{2b}{(n-1)(n-2)} \frac{3}{8} n(n-2) \\
&\leq b\,\mathrm{ld}(n) + l - b\frac{3}{4} \\
&\leq b\,\mathrm{ld}(n)
\end{aligned}
$$

□

The desired complexity of $\mathcal{O}(\log(n))$ follows directly.

5.2.2. Restore_unif and Bend_procedure

Due to its recursive nature, the complexity of the procedures **restore_uniformity/ relocate_pointer** will again be represented by a recurrence relation. But, unlike in the recurrence of **usurp**, the recurrence for **restore_uniformity/relocate_pointer** also depends on the position of *new* in the list. To avoid any problem related to this fact, I resort to the same method as used by Durand in [4]: Consider the insertion of a node in *every* possible position. Dividing by the number of possible insertion positions results in the expected costs for the procedures **restore_uniformity/relocate_pointer**. The argument of the recurrence n will again be the length of the list without its header. Refer to Figure 5.2 for further illustrations.

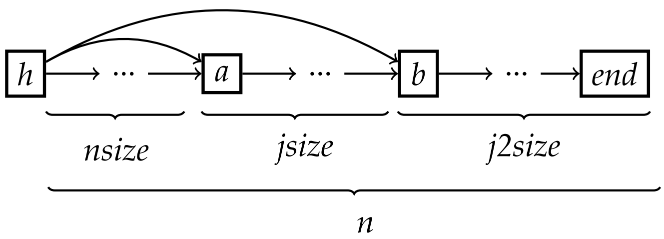


Figure 5.2.: Illustration of the sizes used in equation(5.3).

$$C_{\text{rst}}(2) = C_{\text{usu}}(2) = h_2,\ C_{\text{rst}}(3) = C_{\text{usu}}(2) + C_{\text{spec}} = h_3,$$
$$C_{\text{rst}}(4) = C_{\text{usu}}(3) + C_{\text{rst}}(2) + C_{\text{rst}}(2) = h_4 \qquad h_2, h_3, h_4 \in \mathbb{R}^+$$

$$\begin{aligned} C_{\text{rst}}(n) &= n\frac{2}{n-1}C_{\text{reb}}(n) + (1 - \frac{2}{n-1})\frac{1}{\binom{n-2}{2}}\Bigg[\sum_{i=2}^{n-3}\sum_{j=1}^{n-2-i} C_{\text{usu}}(i+1) \\ &\quad + \sum_{i=2}^{n-3}\sum_{j=1}^{n-2-i} C_{\text{rst}}(i) \sum_{i=1}^{n-3}\sum_{j=2}^{n-2-i} C_{\text{rst}}(j) + \sum_{i=1}^{n-4}\sum_{j=1}^{n-3-i} C_{\text{rst}}(n-1-i-j)\Bigg] \\ &\leq ln + \frac{1}{\binom{n-1}{2}}\Bigg[\sum_{k=3}^{n-2}(n-1-k)b\,\text{ld}(k) + 3\sum_{k=2}^{n-3}(n-2-k)C_{\text{rst}}(k)\Bigg] \\ &\qquad\qquad (n \geq 5),\ b, l \in \mathbb{R}^+ \end{aligned} \tag{5.3}$$

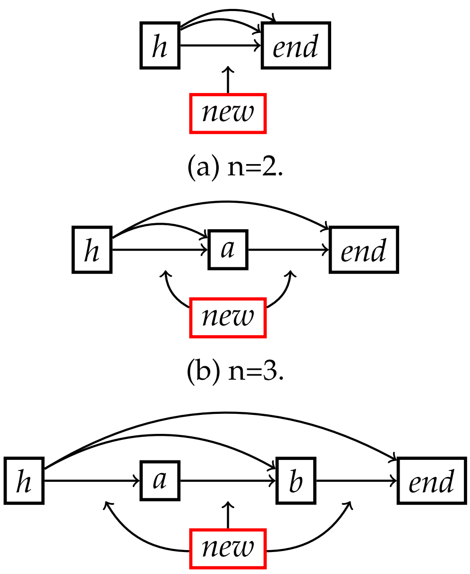
As **usurp**, the algorithm needs to decide whether to bend the pointers of h or not. With probability $\frac{2}{n-1}$, one pointer of h is bent and **rebalance** needs to be called. With n possible positions to insert *new*, this leads to the first summand $n\frac{2}{n-1}C_{\text{reb}}(n) \leq n \cdot l$.

If the algorithm chooses not to bend the pointers, which happens with probability $(1 - \frac{2}{n-1})$, a recursive call needs to be made, depending on the pointers of h and the position of *new*. As every possible insert position needs to be taken into account, equation (5.3) contains a recurrence for all of these positions. The double sums iterates through the sizes of the next- (i) resp. jump-list (j) and the four sums represent the following:

- $\sum_{i=2}^{n-3}\sum_{j=1}^{n-2-i} C_{\text{usu}}(i+1)$ represents the calls of **usurp**. The next-size needs to be at least two, as it was at least one before the insertion ($n-1 \geq 4$).
- $\sum_{i=2}^{n-3}\sum_{j=1}^{n-2-i} C_{\text{rst}}(i)$ represents the recursive calls when *new* is in the next-list. As above, the next-size needs to be at least two after the insertion.
- $\sum_{i=1}^{n-3}\sum_{j=2}^{n-2-i} C_{\text{rst}}(j)$ represents the recursive calls when *new* is in the jump-list. In this case, the jump-size needs to be at least two.

- $\sum_{i=1}^{n-4} \sum_{j=1}^{n-3-i} C_{\text{rst}}(n-1-i-j)$ represents the recursive calls when *new* is in the jump2-list. The jump2-size needs to be at least two, meaning that the next- and jump-sizes need to be smaller than $(n-3)$.

Using symmetry of the double sum and reverse summation, as in equation (4.2), and that the complexity of **usurp** is in $\mathcal{O}(\log(n))$, the final form of equation (5.3) is obtained.
The initial conditions, are illustrated in Figure 5.3.



(a) n=2.

(b) n=3.

(c) n=4.

Figure 5.3.: Initial conditions with possible insertion positions for *new*.

To prove that $C_{\text{rst}}(n) \in \mathcal{O}(n \log(n)$ (I insert at every possible position), I resort to the same method used for $C_{\text{usu}}(n)$:

Lemma 11. *Let* $d = \max\{\frac{h_2}{2\,\text{ld}(2)}, \frac{h_3}{3\,\text{ld}(3)}, \frac{h_4}{4\,\text{ld}(4)}, 4l + \frac{4b}{3}\}$. *Then* $C_{rst}\,(n) \leq dn\,\text{ld}(n)$ *for* $n \in \mathbb{N}_{\geq 2}$.

Proof. The proof is conducted by induction.

Induction Beginning:

$$\begin{aligned} n = 2: &\quad C_{\text{usu}}(2) = h_2 \leq d2\,\text{ld}(2) \\ n = 3: &\quad C_{\text{usu}}(3) = h_3 \leq d3\,\text{ld}(3) \\ n = 4: &\quad C_{\text{usu}}(4) = h_4 \leq d4\,\text{ld}(4) \end{aligned}$$

Induction Hypothesis:

For an arbitrary fixed number n, holds for all $k \in \mathbb{N}$ with $k < n$ that $C_{\text{rst}}(k) \leq dk\,\text{ld}(k)$.

Induction Step:

Using that $\text{ld}(x) \leq x - 1$ for $x > 1$ and the same estimation for $\text{ld}(n) - \text{ld}(k)$ as in the proof of Theorem 10, it holds for $n \geq 5$ that:

$$
\begin{aligned}
C_{\text{rst}}(n) &\leq ln + \frac{1}{\binom{n-1}{2}}\left[\sum_{k=3}^{n-2}(n-1-k)b\,\text{ld}(k) + 3\sum_{k=2}^{n-3}(n-2-k)C_{\text{rst}}(k)\right] \\
&\overset{\text{IV}}{\leq} ln + \frac{2}{(n-1)(n-2)}\left[\sum_{k=3}^{n-2}(n-1-k)b\,\text{ld}(k) + 3\sum_{k=2}^{n-3}(n-2-k)dk\,\text{ld}(k)\right] \\
&\leq ln + \frac{2b}{(n-1)(n-2)}\sum_{k=3}^{n-2}(n-1-k)(k-1) \\
&\quad + \frac{6d}{(n-1)(n-2)}\sum_{k=2}^{n-3}(n-2-k)(k\,\text{ld}(n) - (k\,\text{ld}(n) - k\,\text{ld}(k)) \\
&= ln + \frac{2b}{(n-1)(n-2)} \cdot \frac{(n+1)(n-3)(n-4)}{6} \\
&\quad + \frac{6d}{(n-1)(n-2)}\left[\sum_{k=2}^{n-3}(n-2-k)k\,\text{ld}(n) - \sum_{k=2}^{n-3}(n-2-k)k(\text{ld}(n) - \text{ld}(k))\right] \\
&\leq ln + \frac{b}{3}(n+1) + \frac{6d\,\text{ld}(n)}{(n-1)(n-2)} \cdot \frac{(n+1)(n-3)(n-4)}{6} \\
&\quad - \frac{6d}{(n-1)(n-2)}\sum_{k=2}^{n/2}(n-2-k)k1 \\
&= ln + \frac{b}{3}(n+1) + \underbrace{\frac{(n+1)(n-3)(n-4)}{(n-1)(n-2)}}_{\leq n \text{ for } (n \geq 5)} d\,\text{ld}(n) \\
&\quad - \frac{6d}{(n-1)(n-2)} \cdot \frac{(n-4)(n-2)(2n+9)}{24} \\
&\leq dn\,\text{ld}(n) + ln + \frac{b}{3}(n+1) - d\frac{(n-4)(2n+9)}{4(n-1)} \\
&\leq dn\,\text{ld}(n)
\end{aligned}
$$

because

$$
\begin{aligned}
& ln + \frac{b}{3}(n+1) - d\frac{(n-4)(2n+9)}{4(n-1)} \leq 0 \\
\iff & ln + \frac{b}{3}(n+1) \leq d\frac{(n-4)(2n+9)}{4(n-1)} \\
\iff & 4l \underbrace{\frac{n(n-1)}{(n-4)(2n+9)}}_{\leq 1 \text{ for } (n \geq 5)} + \frac{4b}{3} \underbrace{\frac{(n+1)(n-1)}{(n-4)(2n+9)}}_{\leq 1 \text{ for } (n \geq 5)} \leq d.
\end{aligned}
$$

□

The desired complexity of $\mathcal{O}(n \log(n))$ follows directly.

Corollary 3. *The procedures* ***restore_uniformity/relocate_pointer*** *have an expected runtime complexity of* $\mathcal{O}(\log(n))$.

6. Conclusion

In this thesis, I present an extended form of jumplist defined by Brönnimann, Cazals and Durand in [1]. Instead of only allowing one jump-pointer per node, every node possesses two, with the jump2-pointer reaching even further ahead in the list.

This change speeds up the search, as the choice between two jump-pointers allows a more precise navigation through the list. As seen in Theorem 6, the expected internal path length of a 2-jumplist has a prefactor $\frac{6}{5}$ on the leading term, opposed to the prefactor 2 (Theorem 1 in [1]) of the regular jumplist. Similarly, the expected number of comparisons during a search for every node in the list has a prefactor $\frac{12}{5}$ on the leading term (see Theorem 9), opposed to a prefactor of 3 in Theorem 2 ([1]).

Despite the more complex structure of the 2-jumplist compared to the regular jumplist, the complexity of generation and insertion remains linearithmic.

For further work, even more jump-pointers could be added to every node, to make the search faster. To speed up the choice for the right jump-pointer to follow, a binary search could be used on the jump-pointers. But at some point, the costs for the choice of the right jump-pointer would exceed the savings from the additional jump-pointers, even using binary search.

To avoid this problem, the number of jump-pointers of a node could be adjusted to the length of its sublist, to ensure that every additional jump-pointer enhances the performance. As the size of the sublist for each node is known during generation and insertion, this modification seems to be feasible.

Bibliography

[1] Hervé Brönnimann, Frédéric Cazals and Marianne Durand *Randomized Jumplists: A Jump-and-Walk Dictionary Data Structure.* In H. Alt and M. Habib (Eds.): STACS 2003, LNCS 2607, pp. 283-294, 2003.

[2] Sebastian Wild *Java 7's Dual Pivot Quicksort.* Master Thesis (2012), available at http://wwwagak.cs.uni-kl.de/sebastian-wild.html.

[3] Sebastian Wild, Markus E. Nebel and Ralph Neininger *Average Case and Distributional Analysis of Dual Pivot Quicksort.* ACM Transactions on Algorithms 11, 3, Article 22 (January 2015).

[4] Marianne Durand *Combinatoire analytique et algorithmique des ensembles de donnés.* PhH Thesis (2004), available at http://algo.inria.fr/durand/.

[5] Ronald L. Graham, Donald E. Knuth, Oren Patashnik *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ,1994. ISBN:0201558025.

[6] Amr Elmasry *Deterministic Jumplists.* Nordic Journal of Computing 12, 27-39 (2005).

[7] Luis Barba and Pat Morin *Top-Down Skiplists.* arXiv:1407.7917.

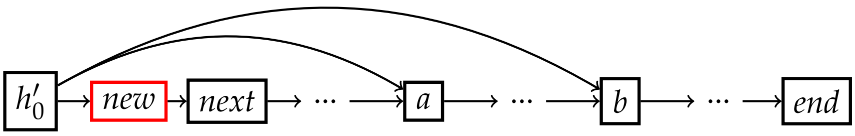
[8] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, Charles E. Leiserson *Introduction to Algorithms.* MIT Press, 3rd edition, 2009. ISBN 978-0-262-03384-8..

[9] Bose, Prosenjit, Karim Douïeb, and Pat Morin *Skip lift: A probabilistic alternative to red-black trees.* Journal of Discrete Algorithms 14 (2012): 13-20.

Appendices

A. Supplementary Proofs

Conditions of Correctness in the Induction Beginning of Theorem 3



- **(c1):** As the pointers of h'_0 are not altered (otherwise h'_0 would be the last node considered by **usurp**) and every node in the list fulfilled the conditions of correctness before the insertion, the pointers do not cross afterwards.
- **(c2):**
  - **(c2.1):** Due to the insertion of *new*, *next* is now a possible target for the pointers of h'_0. Therefore it needs to be shown that every pair of nodes in the target sublist of h'_0, including *next*, has the same probability to be the target of the pointers of h'_0. This is provided by Lemma 2, setting $h = h'_0$.
  - **(c2.2):** As the sublist of has at least length 4, it must have had at least length 3 before the insertion. Therefore the jump- and jump2-pointer pointed to different nodes before the insertion and as they are not altered also do after the insertion.
  - **(c2.3):** As h'_0 must have had a sublist of length $\geq$3 before the insertion, its pointers did not point to the direct successor before the insertion. The pointers of h'_0 are not altered and thus this also holds after the insertion.

  (c2'): If (1$\leq$ n < 4), a special case would have been applied and h'_0 would have been the last node considered by **usurp**).

Monotony of $C_{\mathbf{gen}}(N)$

To show that $C_{\text{gen}}(N)$ is monotonically increasing, I prove that $C_{\text{gen}}(N+1) - C_{\text{gen}}(N) \geq 0$ for $N \geq 1$.

Proof. The proof is conducted by induction.

Induction Beginning:

- $\mathbf{N = 1}: C_{\text{gen}}(2) - C_{\text{gen}}(1) = c_2 - c_1 \geq 0.$
- $\mathbf{N = 2}: C_{\text{gen}}(3) - C_{\text{gen}}(2) = c_3 - c_2 \geq 0.$
- $\mathbf{N = 3}: C_{\text{gen}}(4) - C_{\text{gen}}(3) = c_4 - c_3 \geq 0.$
- $\mathbf{N = 4}: C_{\text{gen}}(5) - C_{\text{gen}}(4) = (4e + d + \frac{6}{6}(2C_{\text{gen}}(1) + C_{\text{gen}}(2))) - c_4 = 4e + d + 2c_1 + c_2 - c_4 \geq 0.$

Induction Hypothesis:
For a fixed arbitrary number N, holds that for all $k \in \mathbb{N}$ with $k < N$ that $C_{\text{gen}}(k+1) - C_{\text{gen}}(k)$.

Induction Step:

For $N \geq 5$:

$$\begin{aligned}
&C_{gen}(N+1) - C_{gen}(N) \\
&\qquad = eN + d + \frac{6}{(N-1)(N-2)} \sum_{k=1}^{N-2} (N-k-1) C_{\text{gen}}(k) \\
&\qquad\quad - e(N-1) + d + \frac{6}{(N-2)(N-3)} \sum_{k=1}^{N-3} (N-k-2) C_{\text{gen}}(k) \\
&\qquad = e + \frac{6}{(N-1)(N-2)(N-3)} \left[\sum_{k=1}^{N-2} (N-3)(N-k-1) C_{\text{gen}}(k) \right. \\
&\qquad\quad \left. - \sum_{k=1}^{N-3} (N-1)(N-k-2) C_{\text{gen}}(k) \right]
\end{aligned}$$

It is sufficient to show that

$$\left[\sum_{k=1}^{N-2} (N-3)(N-k-1) C_{\text{gen}}(k) - \sum_{k=1}^{N-3} (N-1)(N-k-2) C_{\text{gen}}(k) \right] \geq 0:$$

$$
\begin{aligned}
&\sum_{k=1}^{N-2}(N-3)(N-k-1)C_{\text{gen}}(k) - \sum_{k=1}^{N-3}(N-1)(N-k-2)C_{\text{gen}}(k) \\
&\quad = (N-3)C_{\text{gen}}(N-2) + \sum_{k=1}^{N-3}(N^2-(N-3)k-4N+3)C_{\text{gen}}(k) \\
&\qquad - \sum_{k=1}^{N-3}(N^2-(N-1)k-3N+2)C_{\text{gen}}(k) \\
&\quad = (N-3)C_{\text{gen}}(N-2) + 2\sum_{k=1}^{N-3}kC_{\text{gen}}(k) - N\sum_{k=1}^{N-3}C_{\text{gen}}(k) + \sum_{k=1}^{N-3}C_{\text{gen}}(k) \\
&\quad \overset{(*)}{\geq} (N-3)C_{\text{gen}}(N-2) + (N-2)\sum_{k=1}^{N-3}C_{\text{gen}}(k) - N\sum_{k=1}^{N-3}C_{\text{gen}}(k) \\
&\qquad + \sum_{k=1}^{N-3}C_{\text{gen}}(k) \\
&\quad = (N-3)C_{\text{gen}}(N-2) - \sum_{k=1}^{N-3}C_{\text{gen}}(k) \\
&\quad \geq (N-3)C_{\text{gen}}(N-2) - \sum_{k=1}^{N-3}C_{\text{gen}}(N-2) \\
&\quad = (N-3)C_{\text{gen}}(N-2) - (N-3)C_{\text{gen}}(N-2) = 0
\end{aligned}
$$

At (*), I use Chebyshev's sum inequality: $m \cdot \sum_{i=1}^{m} a_i b_i \geq (\sum_{i=1}^{m} a_i) \cdot (\sum_{i=1}^{m} b_i)$ for $m \in \mathbb{N}_{\geq 1}$ and two increasing (in the same direction) sequences a_i, b_i. As both sequnences are increasing (see **Induction Hypothesis** for $C_{\text{gen}}(k)$), the following estimation can be made:

$$
\begin{aligned}
2\sum_{k=1}^{N-3}kC_{\text{gen}}(k) &\geq \frac{2}{N-3}\Big(\sum_{k=1}^{N-3}k\Big)\cdot\Big(\sum_{k=1}^{N-3}C_{\text{gen}}(k)\Big) \\
&= \frac{2}{N-3}\cdot\frac{(N-2)(N-3)}{2}\sum_{k=1}^{N-3}C_{\text{gen}}(k) \\
&= (N-2)\sum_{k=1}^{N-3}C_{\text{gen}}(k))
\end{aligned}
$$

□