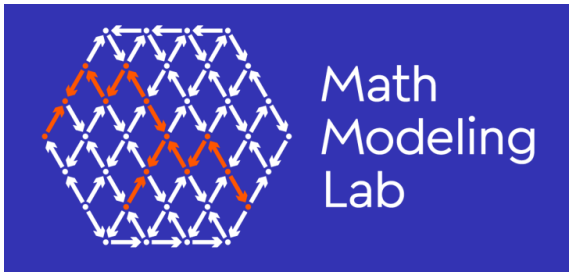
Lectures 5-6. Pointer based data structures

Dmitri Shmelkin

Math Modeling Competence Center at Moscow Research Center of Huawei
Shmelkin.Dmitri@huawei.com

March 23, 2023



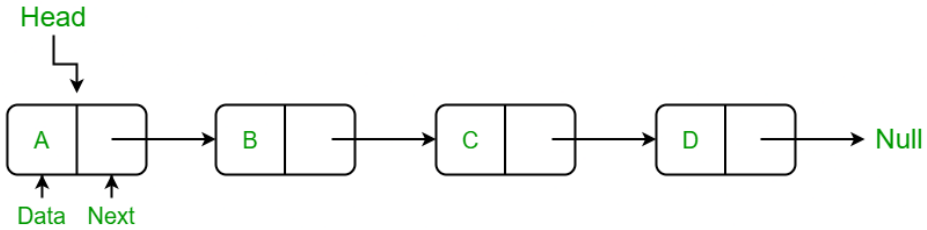
Overview

Recollection and motivation

• In previous lectures we considered structures with a direct addressing, like array and heap that is an array enriched with *Parent* and *Son* relations.
• Then we considered Hash Table that supports a direct addressing with collisions as an issue that can be controlled and handled efficiently.
• However in many cases the direct address may hardly apply, for example, because the structure is dynamically changing from different code lines and the **centralized** management like the doubling trick used in dynamic arrays and dynamic hash tables is not efficient.
• It is also possible that the operations with the stucture are mostly local: For example a FIFO queue in real life does not need global search procedures, and by lack of centralized controller, clients take their position in the queue by just asking "who is the last?" and retaining their **Previous** client info and watching the moment when the Previous becomes the first.
• So like in the above example, it is possible to consider structures with units only having **pointers** to one or several similar units, that is, of oriented graph.
• In this lecture, specific graph classes will be considered and smart algorithms making operations in such structures efficient.

Linked List

Linked list is the simplest of the structures and therefore, the most widely used in code, especially, in legacy C++ code. The unit consists of *Data* field and of a pointer to the *Next* unit. Besides, the user needs to store the *Head* unit as the only method to get access to all them.



This kind of structure is popular due to the flexibility, which is more important, in this case, than standard design. In C++ one typically defines a class of linked list unit with the full user control of *Next*. In such a style, insertion of a data D to the list after some unit LU_0 looks like this:

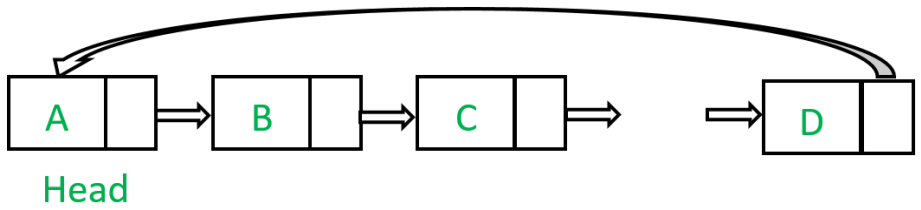
1. Construct a new unit $LU_1 : LU_1 \to Data := D; LU_1 \to Next := LU_0 \to Next$

2. Set $LU_0 \to Next := LU_1$

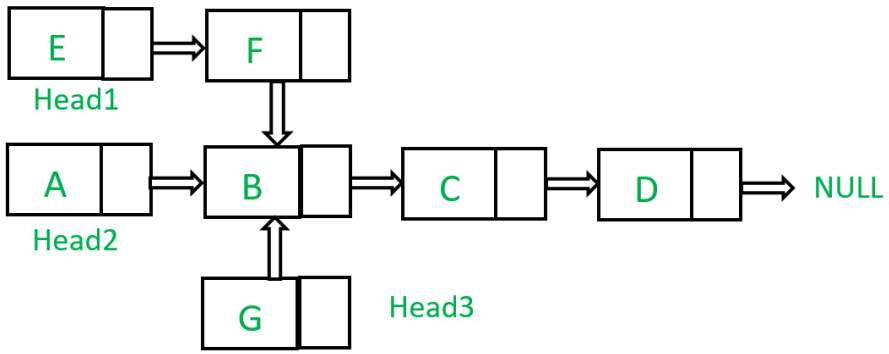
Inserting to the head of list is similar and simpler.

Design Linked List as you wish

To delete an element LU_1 it needs either to be $LU_0 \to Next$ for a known unit LU_0 or to be the head. In the former case we first do:
$LU_0 \to Next := LU_1 \to Next; LU_1 \to Next := NULL$, then safely delete LU_1.
In the latter case: $head := head \to Next; LU_1 \to Next := NULL$, then delete LU_1.
This flexibility allows to design different graphs, e.g., a loop:



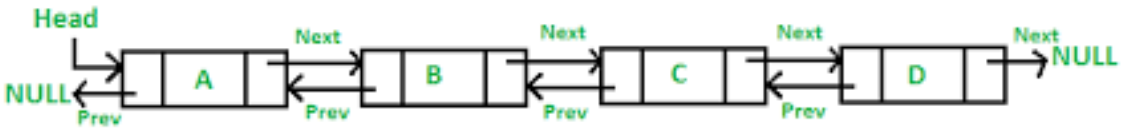
Or even a free mesh. Notice that all source nodes are to be stored as heads:



The **drawback** of Linked List is $O(n)$ complexity of element query.

Bi-directional Linked List

• As we pointed out, the linked list is suitable for the programs, where the access to the elements is local, for example, iteration from head to tail.
• In more subtle cases, several sequential units are processed together, so we might need to return back. In singly linked list it means restart from the head.
• If the width of access "window" is predictable and small, e.g., 3 sequential units. Then a hint is to iterate the topleft and refer to its *Next* and $Next \to Next$.
• In general cases, when the window may be arbitrary, one applies a bidirectional structure, where besides *Data* and *Next* fields, also *Prev* is stored. as on the pic:



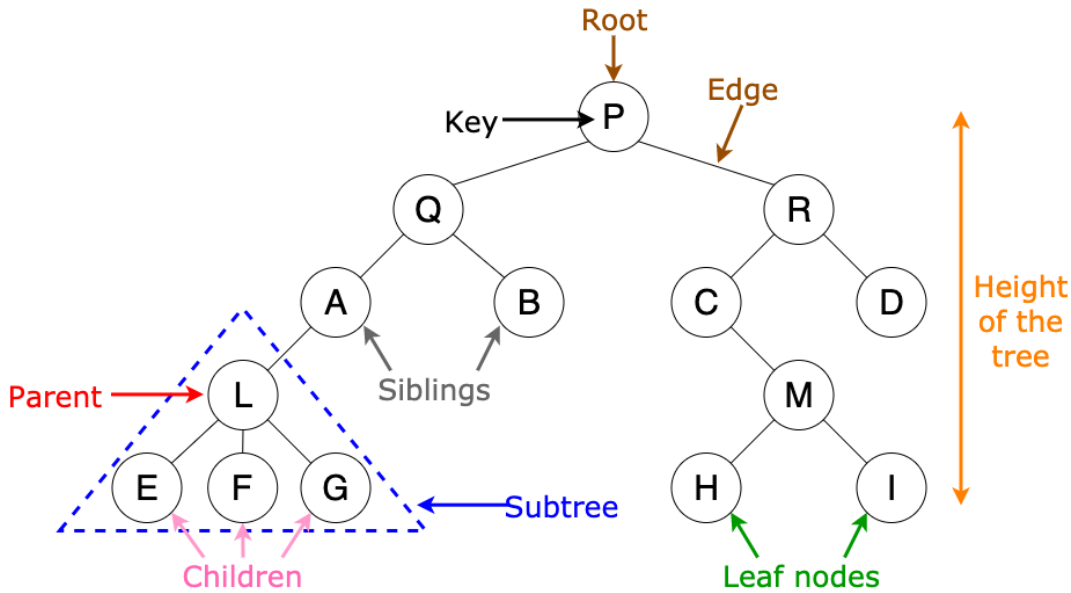
• Operations with bidirectional list are quite similar to those for singly directional one above, just the price of *Insert*/*Delete* is twice.
• However the $O(n)$ complexity of element query is in many cases unaffordable. So we pass to the pointer based structures able to cope with that issue.

Rooted Trees

The word **tree** has several similar senses in discrete math, so to emphasize the meaning of data structure we talk about **Rooted Tree**

Definition 1

An oriented graph is called a *Rooted tree* if it has the only source node and the incoming degree of all other nodes is 1. Every oriented edge goes to the *children* of the node. The data structure is in fact the *double* of the graph such that to every node, its only *parent* is provided as a reverse edge.

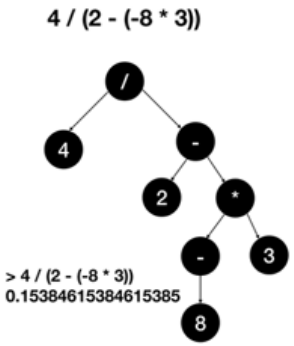


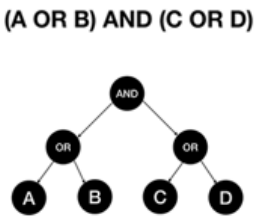
Examples of non-binary trees

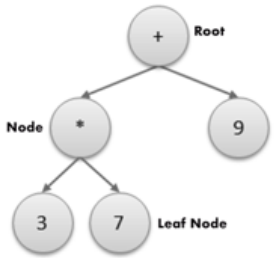
- The rest of the lecture deals with binary trees, the most interesting of all trees
- Non-binary trees are widely used in programming under different specific names:

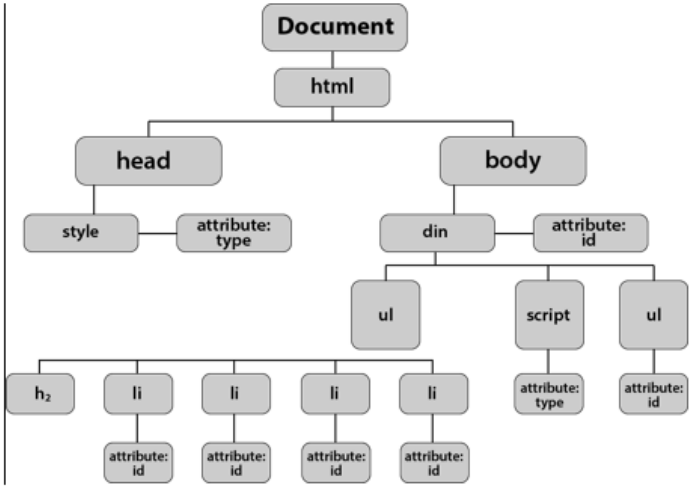
Parenthesis structure arises in many different contexts, from compilers to symbolic computations. In fact, every expression with parenthesis is a tree such that the terms inside a parenthesis are the children of the parenthesis term.

HTML web language defines a document as a non-binary rooted tree structure







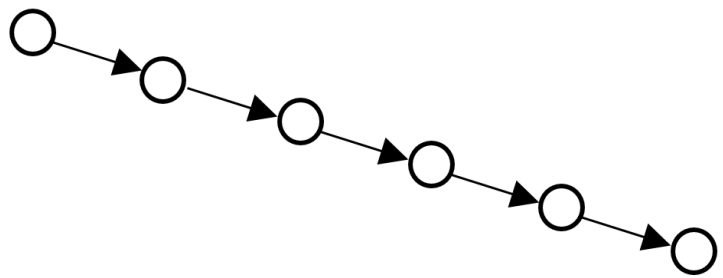


Binary Trees

- A unit of binary tree contains, besides data, pointers to *LeftChild*, *RightChild*, and *Parent*. This makes it similar to a Heap node.
- The node x with $x \to Parent = NULL$ is the root.
- A node with $x \to LeftChild = NULL, x \to RightChild = NULL$ is a leaf.
- Every node is a root of a binary subtree that splits into *left* and *right subtree*
- The shorthest path length from the root to a node is called node's *depth*.
- The maximal depth of a (leaf) node is called *heigth* of the tree.
- A tree is called **balanced** if the height of the left and right subtrees of every node differ by not more than 1.

Lemma 2

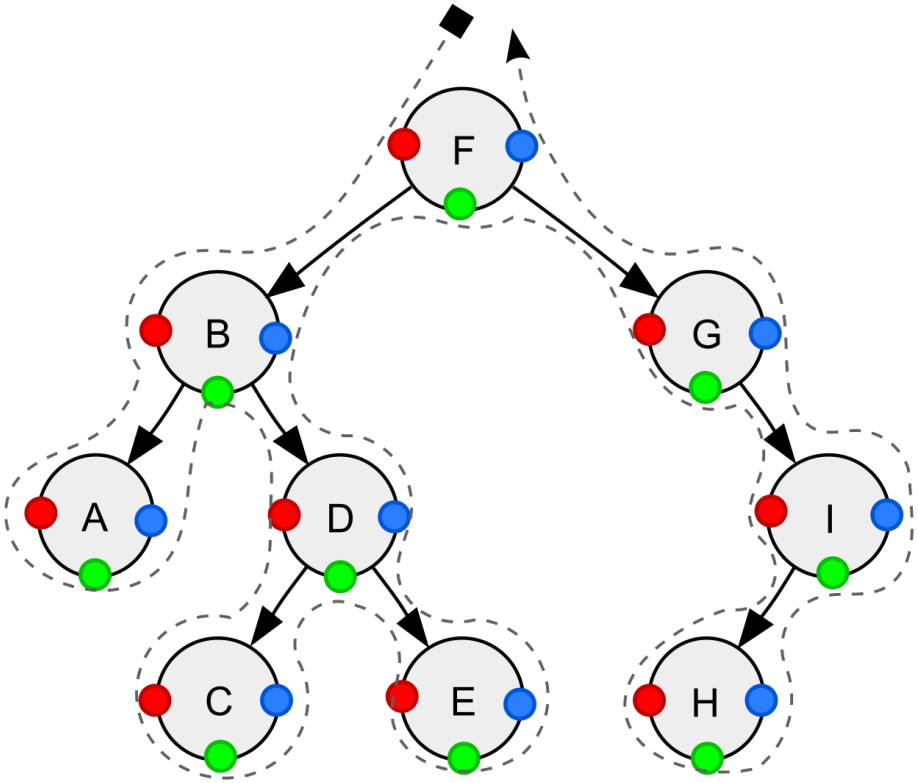
The height of a tree with n nodes varies from $\lfloor \log_2(n) \rfloor$ *for a balanced tree to* $n - 1$ *for a tree similar to a linked list.*



Binary tree traversal

• The main part of binary tree efficiency is due to the recursive **Depth First Search** traversal of the tree, which means, to go recursively deeper to the tree prior to switch to the next child.
• Node's depth is a measure of complexity of visiting the node and the height of the tree is a universal measure of all binary tree operations, as we will see.
• Therefore, by Lemma 2 the cost of operation for balanced tree with n leaves is $O(\log_2(n))$.
• Depending on the application, 3 different modes of traversal are applied:
Pre-order: Also abbreviated as **NLR**, which means, first process the Node itself, then recursively traverse its Left subtree, then the Right subtree.
Post-order: Also abbreviated as **LRN**, Left subtree, Right subtree, Node.
In-order: Also abbreviated as **LNR**, Left subtree, Node, Right subtree.

Binary tree traversal illustration



Assume we apply DFS with each of orders, NLR, LRN, and LNR, and print out the visited node in the appearance order. Then we get:
For Pre-order (NLR): $F, B, A, D, C, E, G, I, H$;
For Post-order (LRN): $A, C, E, D, B, H, I, G, F$;
For In-order (LNR): $A, B, C, D, E, F, G, H, I$.

Binary tree traversal properties

Let's provide a metacode for DFS for just one order, LNR or In-order. Other orders are similar. Assume that the traversal visits all nodes and print these out.
InOrderDFS: Input: a tree node, r. Output: out stream or array of tree keys
1. *if* $(r = NULL)$ *return*
2. $InOrderBFS(r \to LeftChild)$;
3. Print out $r \to Data$;
4. $InOrderBFS(\to RightChild)$;

Theorem 3

InOrderDFS prints every node one time and has complexity $\Theta(n)$*, where* n *is the number of nodes.*

Proof: It is clear from the algorithm that every node is visited one time in line **3**, which implies the complexity statement. The upper complexity also follows from Theorem 9, Lecture 2 about recursion. □

Binary Search Tree definition

Similar to the context of Hash Table, we consider a data structure such that every item has a key - a number (integer or float point), all keys are different, and a structure supports the main functions as follows:

1. Element query or Search: to find the pointer to a tree node with queried key;
2. Finding Maximum and Minimum key present in the tree;
3. Finding the node with the Next (in the ascending order) or Previous key;
4. Insert or Delete a node.

Remark: Notice that, out of our previous smart data structures, Heap is good at either Maximum or Minimum (but not both!), and not good at Query. Hash is good at Query, but not good at operations 2 and 3.
To fulfill operations 1-4 (and many others), we introduce:

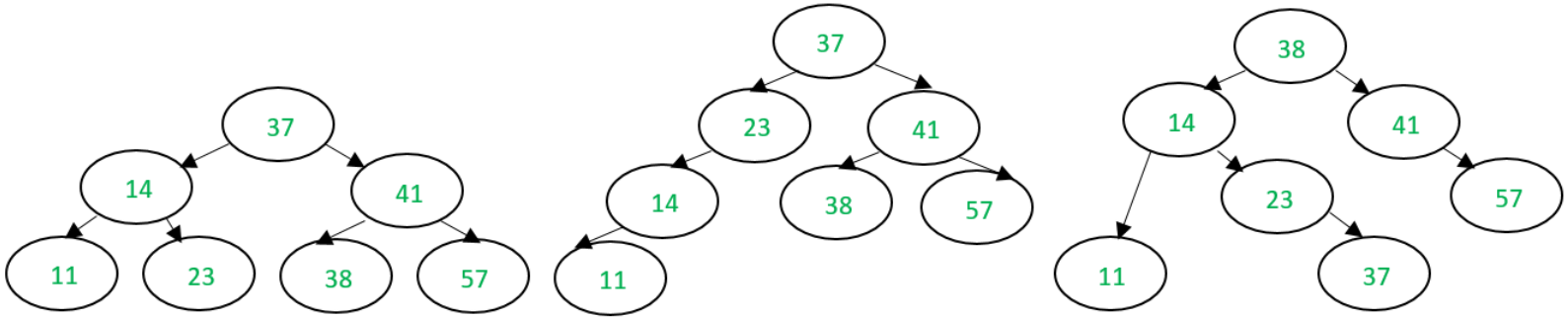
Definition 4

A binary tree is called Binary Search Tree if for every node with key k, all keys in its left subtree are smaller and all keys in its right subtree are greater than k.

Remark: Changing the inequalities in Definition 4 from $<, >$ to $\leq, \geq$ is possible, but makes the operations a bit more complicated.

Binary Search Tree definition: examples

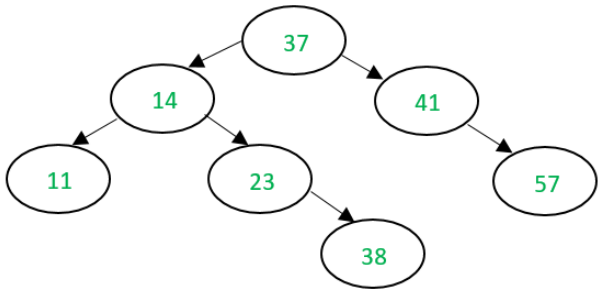
For a given set of keys there can be BST with different structure, which store these keys. For example, consider the set $\{11, 14, 23, 37, 38, 41, 57\}$:



A **local** version of BST definition may look natural: to replace the inequalities for the whole left and right subtrees in Definition 4 with a weaker one:

$$node.LeftChild.key \leq node.key, node.RightChild.key \geq node.key.$$

Is it equivalent? The example shows, it is really weaker, not equivalent:



Building a Binary Search Tree

Given a set K of key how to build a BST out of these numbers? How to ensure it is balanced? Both goals are achieved by the algorithm as follows:

BuildBST: Input: an array K of size n. Output: a BST.

1. Merge Sort the array K in ascending order;

2. Make a root node, r, with the key $K[\lceil n/2 \rceil]$

3. $MakeSubTree(r, \lceil n/2 \rceil - 1, K[1 : \lceil n/2 \rceil - 1], 1]$;

4. $MakeSubTree(r, \lfloor n/2 \rfloor, K[\lceil n/2 \rceil + 1 : n], 0]$;

The above method calls a recursive function:

MakeSubTree: Input: a root tree node r, an integer m, an array L of size m, and Boolean variable *IsLeft*. Output: a BST.

1. Make a sub-tree root node, s, with the key $L[\lceil m/2 \rceil]$

2. $s \rightarrow Parent := r$; *if* $(IsLeft)$ $r \rightarrow LeftChild := s$; *else* $r \rightarrow RightChild := s$;

3. $MakeSubTree(s, \lceil m/2 \rceil - 1, L[1 : \lceil m/2 \rceil - 1], 1]$;

4. $MakeSubTree(s, \lfloor m/2 \rfloor, L[\lceil m/2 \rceil + 1 : m], 0]$;

Properties of BSTBuild algorithm

Theorem 5

BuildBST builds a balanced BST of height $\lfloor \log_2(n) \rfloor$ *in* $O(n \log(n))$.

Proof: By construction, the binary tree meets Definition 4 property. To prove the height statement, let's make it more precise: the leaves of the output have all depths q, if $n+1 = 2^{q+1}$ and have the depth $\lfloor \log_2(n) \rfloor$ or $\lfloor \log_2(n) \rfloor - 1$, otherwise. We prove the latter statement by induction. First of all, if $n = 2^{q+1} - 1$, then recursive steps deal with $m = 2^q - 1, 2^{q-1} - 1, \ldots, 1$. Otherwise, we always split a tree with n node into a root node and 2 subtrees with $\lceil n/2 \rceil - 1$ and $\lfloor n/2 \rfloor$ nodes. Applying the induction step, each subtree has 1 or 2 values of leaves depth, and eiher both subtrees have 2 values and the pairs are equal or one subtree has one value and this value is one of the values for another subtree. In both cases, the tree height increases at 1 on recursive step. So we got both the formula for the height and that the tree is balanced. As for the complexity, by Theorem 12, Lecture 2, Merge Sort takes $O(n \log(n))$ and the recursion takes $O(n)$ by Theorem 9, Lecture 2.

DFS traversal in BST and low bound of building a BST

Theorem 6

Applying InOrderDFS to a BST, the output is that we print out the keys in the ascending order, with complexity $\Theta(n)$

Proof: By the BST property in Definion 4, and due to LNR order, we always print out smaller values before larger ones. The complexity is given by Theorem 3. □

Corollary 7

Building BST out of an array of key belongs to $\Theta(n\log(n))$.

Proof: If building a BST, then applying an $\Theta(n)$ BFS, we get a sorted array, so building BST is not of smaller complexity than sorting, which is in $\Omega(n\log(n))$ by Theorem 17, Lecture 2. As for upper bound, By Theorem 5, Building BST belongs to $O(n\log(n))$. □

BST subtrees are intervals

Corollary 8

Let q be a node in a BST, then all keys in the subtree with root q constitute an interval of keys in the ascending order. More precisely, let k_{min} and k_{max} be the minimal and the maximal key in the subtree. Then the subtree contains all keys of the whole tree from the interval $[k_{min}, k_{max}]$.

Proof: By Theorem 6, LNR DFS prints out the keys in the ascending order. When DFS reachs q it will print out the keys of the subtree, hence, these constitute an interval. □

BST operations: Search

BSTSearch: Input is a BST with root r and a key k to be found. Output is either the pointer to the node with required key or *NULL*, if not found.

1. *if* $(r = NULL$ *or* $r \to Key = k)$
2. *return* r
3. *if* $(k < r \to Key)$ *return* $BSTSearch(r \to LeftChild, k)$
4. *else return* $BSTSearch(r \to RightChild, k)$

Remark: 1. *BSTSearch* is *similar to* a pre-order DFS, NLR.
2. *BSTSearch* finds the node if the key is present and it uses node's depth steps to get it. If the key is not present, the routine reaches a leaf m, then m has no children and line **2** returns *NULL*. So $BSTSearch \in O(h)$, where h is the height.
3. The leaf m in **2** has the key that is either left end or right end of the interval containing k out of the minimal intervals between the keys out of the tree (cf. Range Search, Lecture 1.)

An iterative version of Search function is also simple:

1. *while* $(r \neq NULL$ *and* $r \to Key \neq k)$ *do*
2. *if* $(k < r \to Key)$ $r := r \to LeftChild$;
3. *else* $r := r \to RightChild$;
4. *return* r

BST operations: Maximum and Minimum

Finding the Maximum with $O(1)$ compexity is the specific advantage of Heap that is built for this purpose, but Max-Heap is unable to report the Minimum, so another heap is needed for this task. BST is able to support both Maximum and Minimum functions with $O(h)$ complexity:

BSTMin Input is a BST root node, r. Output: the node with the maximal key

1. *while* $(r \neq NULL$ *and* $r \rightarrow LeftChild \neq NULL)$ *do*

2. $\quad r := r \rightarrow LeftChild;$

3. *return* r

Remark: The algorithm is based on the fact that the node containing the minimal key is leftmost, so to get it, the method just descends from the root down to the left children. Analogously, the Maximum is obtained by descending to the right and the code of it is similar.

BST operations: Next and Previous keys

Definition 9

For a BST node, n, except for the Maximum, let $Next(n)$ be the node with the key going after $n \to key$, in the ascending order. Similarly, define $Prev(n)$.

Lemma 10

1. *If n has a right child, then $Next(n)$ is the Minimum of right subtree of n.*
2. *If $n \to RightChild = NULL$, then consider the path from n to root r: $n_0 = n, n_1 = n_0 \to Parent, n_2 = n_1 \to Parent, \ldots, n_d = n_{d-1} \to Parent = r$. Then either n is the Maximum or $Next(n) = n_j$ such that $n_{j-1} = n_j \to LeftChild$ and j is the minimal with such property.*

Proof: 1. By Corollary 8, $Next(n)$ in the subtree of n, if exists, is that in the whole tree. And if n has right child, then the Minimum of right subtree is the smallest key under n larger than $n \to Key$. **2.** If n is not the whole tree Maximum, then, it is not rightmost, hence, the node n_j exists and unique. Since n is in the left subtree of n_j, $n_j \to Key > n \to Key$ and n is the maximum in the left subtree of n_j, so by **1** and Corollary 8, $n = Prev(n_j)$, hence, $n_j = Next(n)$.

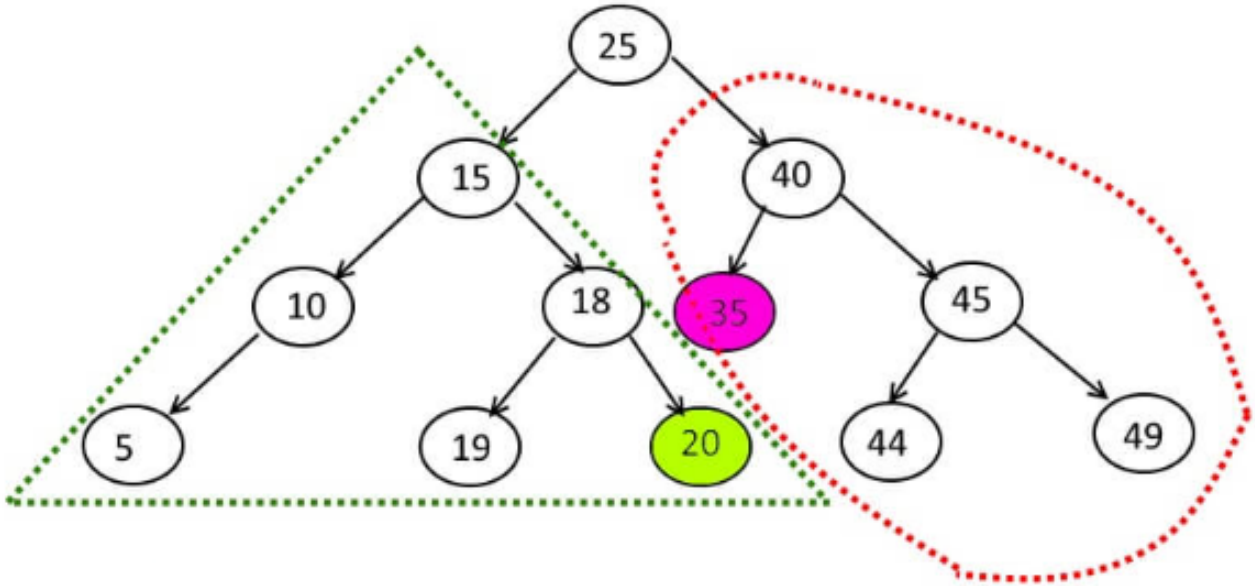
BST operations: Next and Previous keys code

By Theorem 10, an algorithm for *Next* is as follows:

BSTNext: Input: a node n. Output: $Next(n)$ or $NULL$ if n is Maximum.

1. *if* $(n \to RightChild \neq NULL)$ *return* $BSTMin(n \to RightChild)$;
2. $p := n \to Parent$
3. *while* $(p \neq NULL$ *and* $n = p \to RightChild)$
4. $\quad n := p$; $p := p \to Parent$;
5. *return* p

The code of *BSTPrev* is similar.



BST operations: Insert a key

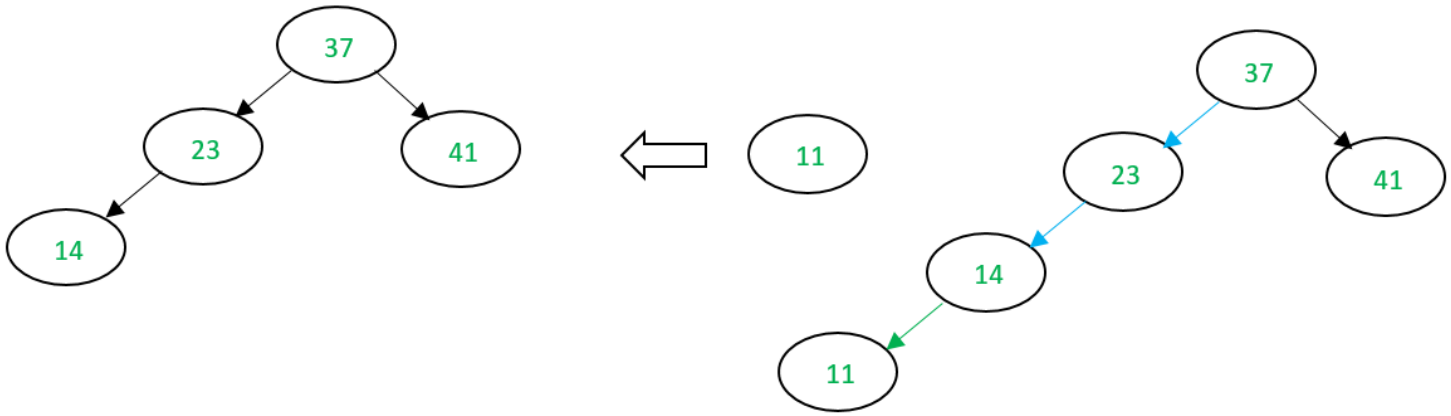
BSTInsert: Input: the root node, r and a key, k. Assumptions: $r \neq NULL$ and k is not present in the tree. Output: the tree contains a new node with key k.

1. Make a node q with $q \to Key = k$ and $NULL$ other fields values;
2. $p := NULL; n := r;$ //p is always the parent of n
3. *while* $(n \neq NULL)$ *do*
4. $p := n;$//move p down
5. *if* $(k < n \to Key)$
6. $n := n \to LeftChild;$ //move n down to the left
7. *else* $n := n \to RightChild;$ //move n down to the right
8. $q \to Parent := p$//$n = NULL$ means q is to be p's child
9. *if* $(k < p \to Key)$
10. $p \to LeftChild := q;$
11. *else* $p \to RightChild := q;$

Remark: Like in several previous cases, *Insert* to a data structure goes the routine of *Search* and the point, where *Search* comes to negative result becomes the parent for the new node.

BSTInsert example

Consider an example of adding key 11 to a balanced BST:



In particular, after insertion the tree is not balanced anymore.

BST operations: Delete a node

Deleting a node n needs to ensure the tree meets Definition 4 property after deletion. It needs nothing, if n was a leaf. In the second case, if n had just one child, it only needs to link this child correctly to $n \to Parent$. The third general case seems tricky but it reduces to the second one thanks to Lemma 11 below.

BSTDelete: Input is the node, n of a tree with root r. Output: n is removed and the tree is a BST. Assumption: the tree does not consist of just $n = r$.

1. $p := n \to Parent$;//$p = NULL$ is possible, if n is the root
2. if ($n \to LeftChild = NULL$ and $n \to RightChild = NULL$) //first case, a leaf
3. if ($n = p \to LeftChild$) $p \to LeftChild := NULL$;
4. else $p \to RightChild := NULL$;
5. else if ($n \to LeftChild = NULL$) //second right case
6. if ($p = NULL$) //remove the root with only right subtree
7. $r := r \to RightChild$; $r \to Parent := NULL$;
8. else if ($p \to LeftChild = n$)
9. $p \to LeftChild := n \to RightChild$; $n \to RightChild \to Parent := p$;
10. else $p \to RightChild := n \to RightChild$; $n \to RightChild \to Parent := p$;
11. else if ($n \to RightChild = NULL$) //second left case
12-16. Symmetrical to **6-10** just replace $n \to RightChild$ with $n \to LeftChild$

BST operations: Delete a node, third case

//here we are with third case, when n has both left and right children

Lemma 11

If n has both left and right children, then holds: $Next(n) \neq NULL$ and $Next(n) \rightarrow LeftChild = NULL$.

Proof: By Lemma 10.1, $Next(n)$ is the Minimum of the right subtree of n and by a Remark after *BSTMin*, a Minimum of a subtree has no left child. □

//now we finish the third case of BSTDelete:

17. $q := Next(n)$;

18. $n \rightarrow Key := q \rightarrow Key$; //instead of n we will delete q

//at this point n and q have the same key, formally violating BST property. This is no problem because we might set $q \rightarrow Key := q \rightarrow Key + \varepsilon$ and get BST

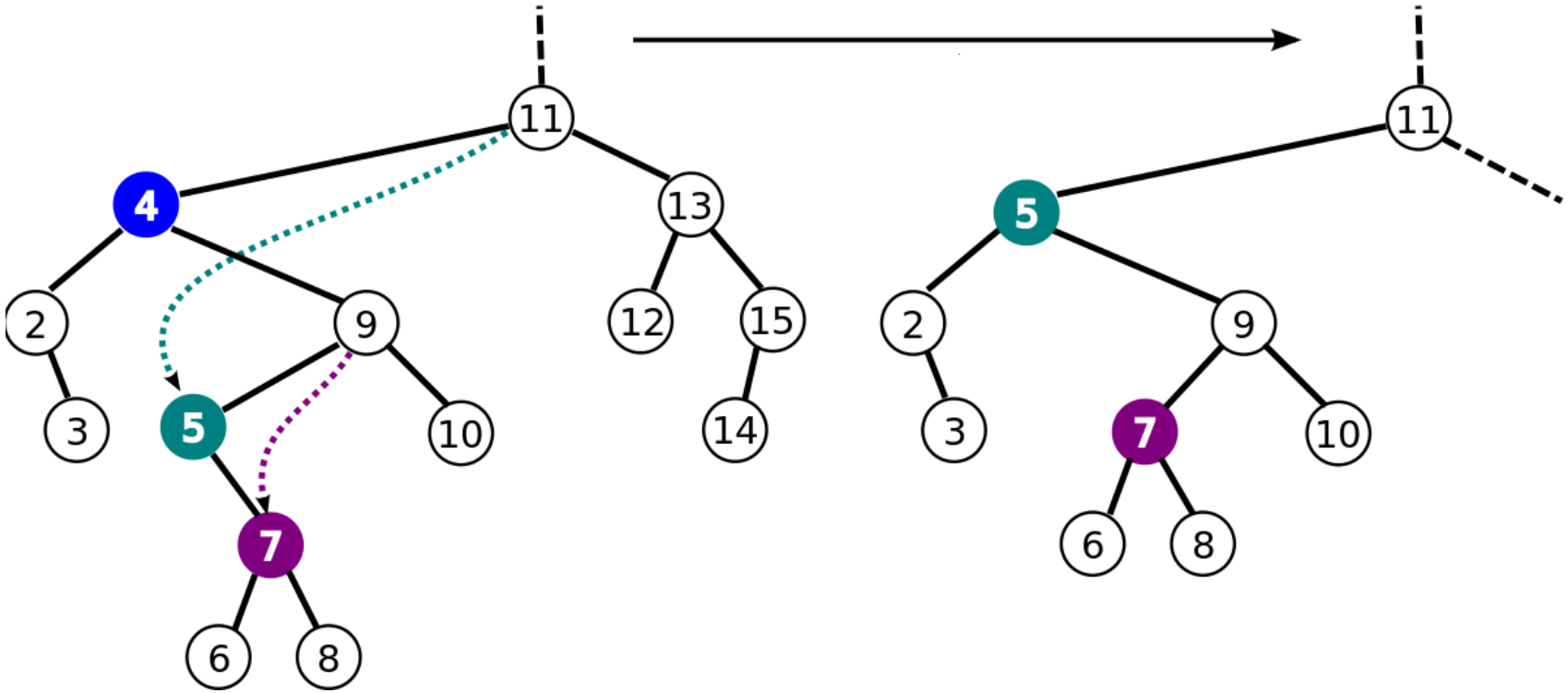
19. $n := q; p := n \rightarrow Parent$//reduce to second right case, line **5**

20-22. Copy of **8-10** /$n = q$ is not the root, so **6-7** are skipped

23. delete n;//all pointers between n and BST are vanished, safe to delete

BSTDelete a node, third case case illustration

Consider an example of *BSTDelete* third case, deleting the node n with key 4



- We find node with key 5 as $Next(n)$.
- As Lemma 11 states, $Next(n)$ has no left child.
- We then migrate key 5 to n, then
- Link the node with key 7, the child of that with 5, to the parent with 9.

Summary of Binary Search Tree: efficiency

Theorem 12

For a Binary Search Tree of height h the functions have all $O(h)$ complexity: BSTSearch, BSTMin, BSTMax, BSTNext, BSTPrev, BSTInsert, BSTDelete.

Proof: It can be checked by direct inspection of each method. Notice also that some methods call other ones like, e.g., *BSTNext* calls *BSTMin* and *BSTDelete* calls *BSTNext*. □

• By Lemma 2, the height h is in $O(\log(n))$, if the tree is balanced, so in this case all the above BST functions are quite efficient.

• If building the tree from a set of keys by *BuildBST* method, then, by Theorem 5 the tree is indeed balanced. But if we assume the set of keys to be dynamic and utilize *BSTInsert* and *BSTDelete*, then we have no guarantees that the tree remains balanced.

• A reasonable approach is to do periodic balancing of the tree when the height gets larger than $\log(n)$. In particular, it can be done with the help of an in-place $O(n)$ Day-Stout-Warren algorithm, [Stout-Warren 86].

• Alternatively, one may use some additional structures on the top of BST allowing a low complexity self-balancing algorithms.

Towards automated balancing of the trees

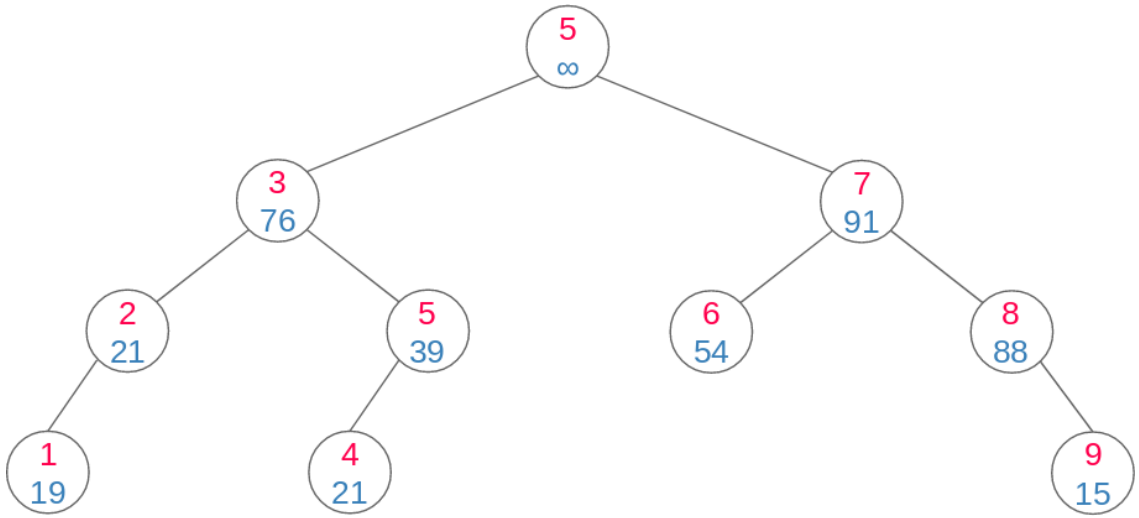
• Probably the most known extension of BST able to efficiently support balancing on *Insert* and *Delete* is so-called **Black-Red Tree**, see [Cormen 3rd ed., Chapter 13] for a detailed introduction to this complicated structure.
• Another well-known approach is to support **AVL tree**, also introduced in [Cormen 3rd ed., ex. 13.3] as a series of exercises. It is amazing that this smart structure has been invented by Soviet mathematicians Georgy Adelson-Velsky and Evgenii Landis in 1962. At that time no Computer Science education was available in USSR and Adelson-Velsky was one of the pioneers of CS. E.Landis is mostly famous for his contributions to PDE theory, pure math.
• In our lecture, we consider an elegant and natural data structure named **Carthesian Tree** ot **Treap**

Carthesian Tree or Treap

The idea of self-balancing is to provide a BST with an additional structure such that preserving this structure on updates implies good chances for BST to be balanced. Recall that a Heap is an array with a binary tree structure that meets heap property: the elements under a node store smaller data than the node itself.

Definition 13

A BST is called *Carthesian Tree* or *Treap* if every node n has, besides the key, $n \to Key$, an additional field named $n \to Priority$ such that, besides BST axiom in Definition 4, holds: the priorities of subtree nodes with route n are less than or equal to $n \to Priority$. Such a BST is similar to a Max-Heap w.r.t. *Priority* field.



Existence and Uniqueness of Treap

Theorem 14

For a set of pairs, $(x_j, y_j), j = 1, \ldots, n$ such that for any $k \neq l$ holds $x_k \neq x_l$, $y_k \neq y_l$, there is a unique structure of treap with nodes $n_j, j = 1 \ldots n$ such that $n_j \to Key = x_j, n_j \to Priority = y_j$. Treap can be built in $O(n \log(n))$ complexity.

Proof: Sorting the pairs, we may assume that $x_1, \ldots, x_n$ is the ascending order. Running from 1 to n, find the index m_0 of the pair with maximal priority,

- By heap property the root holds (x_{m_0}, y_{m_0}) and by BST property its left subtree consists of $(x_j, y_j), j = 1, \ldots, m_0 - 1$, analogously for the right subtree.
- For each of subtrees apply the same idea: select the index m_{00} of the pair with maximal priority out of $y_1, \ldots, y_{m_0-1}$, and m_{01} for the right subtree. Make the nodes $(x_{m_{00}}, y_{m_{00}})$ and $(x_{m_{01}}, y_{m_{01}})$ the left and right children of (x_{m_0}, y_{m_0}) etc.
- In this recursion, the problem is divided into 2 similar ones of total size $n-1$ and the recursive step takes $O(n)$, so Theorem 9 from Lecture 2 yields $O(n \log(n))$ complexity of the recursion. Pre-sorting step is also within this complexity.
- If all priorities are different, then each step has a unique pair with maximal priority, so Treap structure is unique.

Treap is likely to have $O(\log(n))$ depth

Assume that $x_1, \ldots, x_n$ is the ascending order and consider the treap from Theorem 14. For $k < l$ let $Y_{kl} = \{y_k, y_{k+1}, \ldots, y_l\}$ and $Y_{lk} = Y_{kl}$.

Lemma 15

For $k \neq l$, (x_l, y_l) is in sub-treap with root (x_k, y_k) iff y_k is the maximum of Y_{kl}.

Proof: Case 0: Either (x_k, y_k) or (x_l, y_l) is treap's root; this is equivalent to y_k or y_l being global maximum. Assume $(x_m, y_m), m \neq k, l$ is the root. **Case 1:** k and l are neither in left nor in right subtree both. Hence, by BST property, $y_m \in Y_{kl}$ and is the maximum. Also neither of nodes belongs to the subtree below another one. **Case 2:** both nodes are in left sub-treap, apply induction to $(x_j, y_j), j = 1, \ldots, m-1$. Same for right. □

Theorem 16

If $y_j, j = 1, \ldots, n$ are independent homogeneous random variables with the same distribution, then for any $j = 1, \ldots, n$ holds $E(d_j) \in O(\log(n))$, where d_j is the (random) depth of (x_j, y_j) node in the treap from Theorem 14.

Proof of Theorem 16

Proof: Let I_{kl} be the random variable taking 1 if $k \neq l$ and (x_l, y_l) is in the sub-treap with root (x_k, y_k) and 0, otherwise. Then holds: $d_j = \sum_{k=1}^{n} I_{kj}$.
By Lemma 15 and since the indexing does not depend on the priorities:

$$Pr(I_{kl} = 1) = \frac{1}{|k-l|+1}, \text{ if } k \neq l, \text{ and } 0, \text{ otherwise.}$$

$$E(d_j) = \sum_{k=1}^{n} E(I_{kj}) = \sum_{k=1}^{n} Pr(I_{kj} = 1) = \sum_{k=1}^{j-1} \frac{1}{j-k+1} + \sum_{k=j+1}^{n} \frac{1}{k-j+1} \leq \quad (1)$$

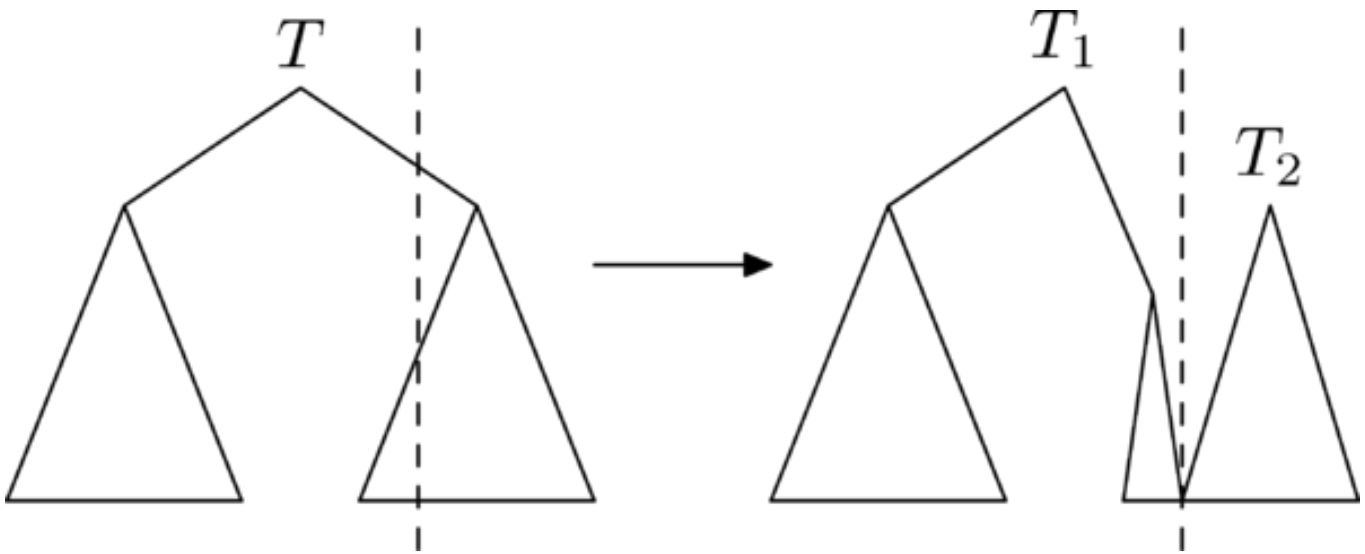
$$\leq \int_2^{j+1} \frac{dt}{t} + \int_2^{n-j+2} \frac{dt}{t} \leq \ln(j+1) - \ln(2) + \ln(n-j+2) - \ln(2) \leq 2\ln(n).$$

Remark: Theorem 16 implies that we can get certain guarantees of Treap height is in $O(\log(n))$ **in average** by assigning random priorities to the keys, taken independently from a homogenous distribution. Then *Search*, *Maximum*, *Minimum*, *Next*, *Previous* will have $O(\log(n))$ complexity, by Theorem 12.
For *Insert* and *Delete* we need to provide methods, **preserving treap structure.**

Split of Treaps

Treap:Split Input: treap T (its root node pointer) and a key value, k. Output: treaps (S, U) such that the keys in S are less than k and those in U are $\geq k$.

1. if $(T = NULL)$ return $(NULL, NULL)$ //we confuse treap with its root pointer
2. if $(T \to Key < k)$ $(T_1, T_2) = Split(T \to RightChild, k)$;
3. $T \to RightChild := T_1$;
4. return (T, T_2);
5. else $(T_1, T_2) = Split(T \to LeftChild, k)$; //$(T \to Key \geq k)$
6. $T \to LeftChild := T_2$;
7. return (T_1, T);

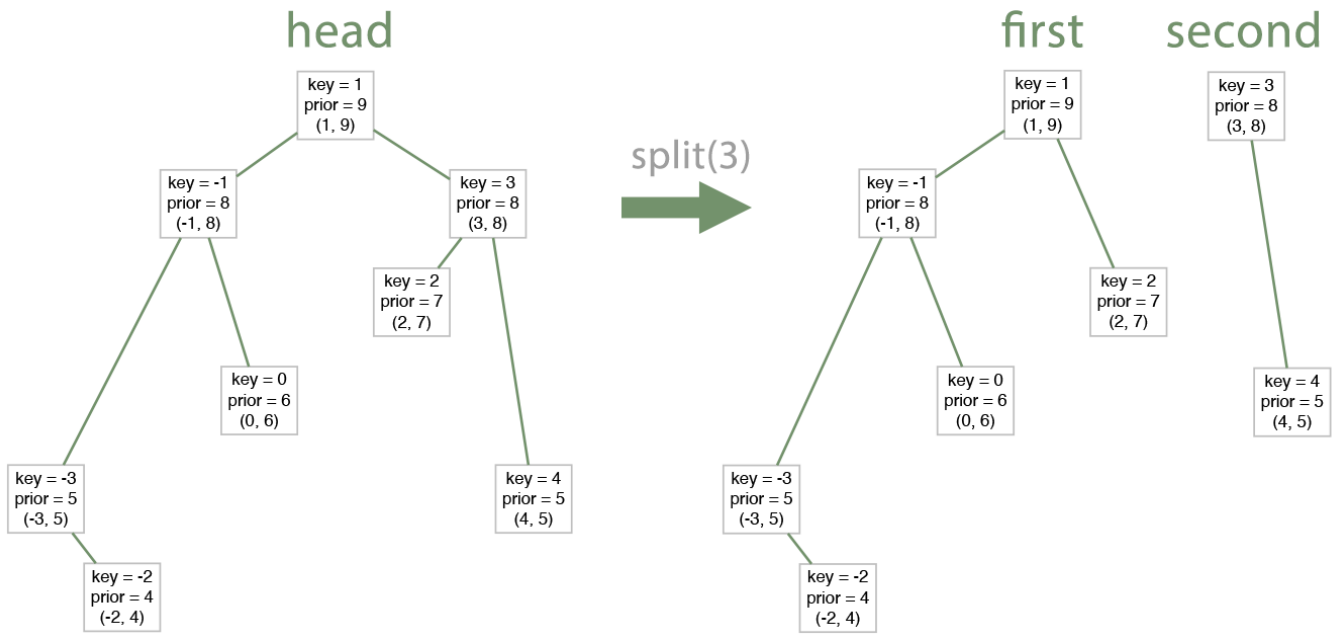


Split of Treaps: properties

Theorem 17

a. *Split is applicable to any BST T. For a treap T it returns 2 treaps.*
b. *Split is in $O(h)$, where h is the height of T.*

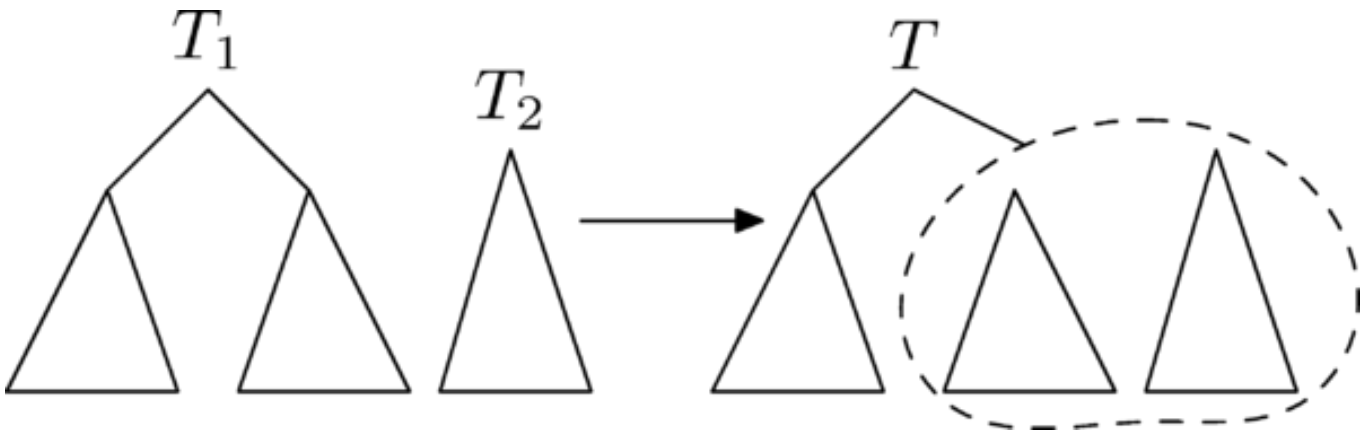
Proof: a: We see that we don't use priorities in the algorithm. Recursively, we prove that both S, U are BST. Indeed, assuming this is true for *Split* output in **2**, then we attach a BST T_1 with keys $> r \to Key$ to r. Same for **5,6**. That S, U fulfill Treap definition is obvious. **b:** *Split* does $\leq h$ steps, each in $O(1)$. □



Merge of Treaps

Treap:Merge: Input is a pair of treaps, T_1, T_2 such that $Max(T_1) < Min(T_2)$. Output is a treap with the union of nodes from both treaps.

1. if $(T_1 = NULL)$ return T_2
2. if $(T_2 = NULL)$ return T_1
3. if $(T_1 \rightarrow Priority > T_2 \rightarrow Priority)$
4. $\quad T_1 \rightarrow RightChild := Merge(T_1 \rightarrow RightChild, T_2)$;
5. $\quad$ return T_1
6. else $T_2 \rightarrow LeftChild := Merge(T_1, T_2 \rightarrow LeftChild)$;
7. $\quad$ return T_2

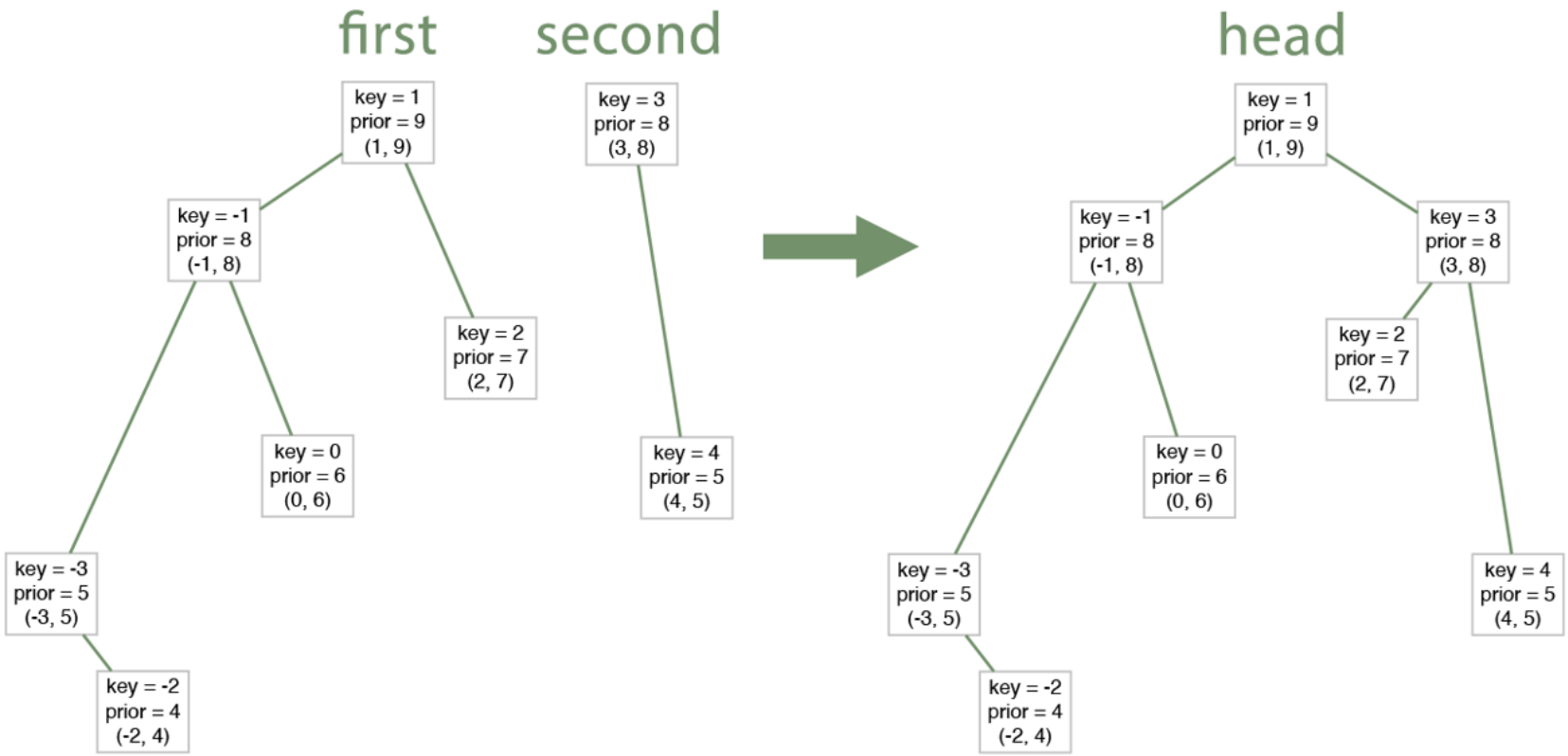


Merge of Treaps: properties

Theorem 18

Merge of treaps returns a treap in $O(h)$ complexity.

Remark: Priorities are used in *Merge* to get a treap and to identify uniquely, which of two roots should dominate. For a general BST a similar algorithm works with an arbitrary selection of dominating root at every step.



Insert/Delete a node to Treap

Treap:Insert Input is a treap T and a new treap node n without parent and children. Output: the treap with node inserted.

1. $(T_1, T_2) = Split(T, n \to Key)$;//to apply *Merge*, need *Split*
2. $T_1 = Merge(T_1, n)$;//*Merge* applies because the keys of T_1 are $< n \to Key$
3. $T = Merge(T_1, T_2)$; return T

Treap:Delete Input is a treap T and a node n in it to be deleted. Output is the treap without n.

1. $(T_1, T_2) = Split(T, n \to Key)$;
2. $n_1 = Minimum(T_2)$; //$n_1 \to Key = n \to Key$ after *Split*
3. $Remove(T_2, n_1)$; //n_1 is a leaf, just vanish its parent's child.
4. $T = Merge(T_1, T_2)$; return T

Theorem 19

Treap : Insert and Treap : Delete work correctly in $O(h)$ time, where h is the height of the treap.

Binary Search Tree Conclusions

• BST is the most universal and flexible data structure to store keys in a dynamic container with an efficient element-by-key query, finding maximum and minimum, the next and previous keys, and many other more specific queries.
• BST allows for efficent *Insert* and *Delete* and these operations as well as the above queries have all $O(h)$ complexity, where h is the height of the tree.
• A balanced tree height is $O(\log(n))$, where n is the number of nodes stored.
• But in the worst case a tree degenerates to a linked list, with $O(n)$ height.
• For a given set of keys, one builds a balanced BST using $O(n \log(n))$ complexity. However, standard *Insert* and *Delete* do not preserve balancedness.
• There are some approaches to enable BST node with additional data and use it to preserve balancedness on updates, such as Black-Red or AVL trees. Both are popular and BR tree is used in standard containers, but is it quite complicated.
• We consider another similar approach, Treap, which is quite natural and gurantees logarithmic average node depth, if priorities are selected at random.
• *Insert* and *Delete* of a treap are implemented via *Split* and *Merge*, and the latter methods generalize to any BST. These methods preserve treap structure and therefore, the average balancedness, due to the priorities selected.

References

Q.F. Stout and B.L. Warren, *Tree rebalancing in optimal time and space*, *Communications of the ACM*, 1986.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, third edition 2009.