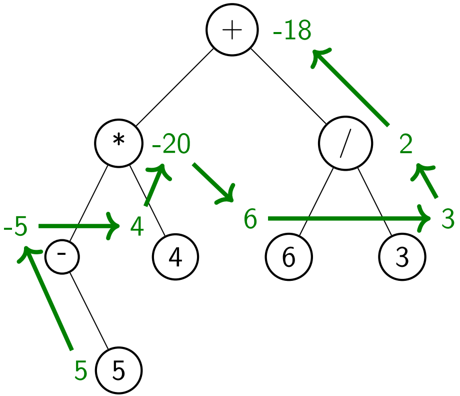
Topic 5.4

Tree walks

Application : Evaluating an expression

Example 5.13

If we want to evaluate an expression represented as a binary tree, we need to visit each node and evaluate the expression in a certain order.



In green, we have evaluated the value of the node. The path indicates the order of evaluation.

Tree walks

Visiting nodes of a tree in a certain order are called tree walks.

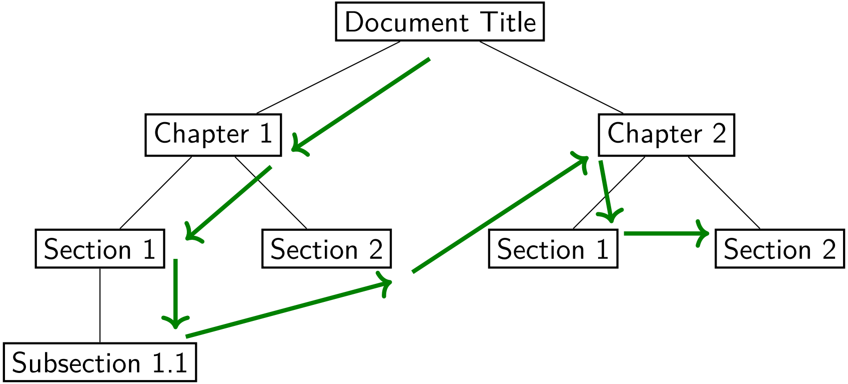
There are two kinds of walks for trees.

- preorder: visit parent first
- postorder: visit children first

Example: preorder

Example 5.14

Let a document be stored as a tree. We read the document in preorder.



Preorder/Postorder walk

Algorithm 5.1: PreOrderWalk(n)

```
visit(n);
for n' ∈ children(n) do
    PreOrderWalk(n');
```

Algorithm 5.2: PostOrderWalk(n)

```
for n' ∈ children(n) do
    PostOrderWalk(n');
visit(n);
```

The first example of expression evaluation is postorder walk.

Commentary: visit(v) is some action taken during the walk.

Walking on ordered tree

How do we walk on an ordered tree?

For an ordered tree, we may visit children in the given order among siblings.

We may have choices to change the order of visits among ordered siblings.

Commentary: Our algorithm works for both ordered and unordered trees. Our algorithm does not specify the order of visits of siblings for unordered trees. Please pay attention to the subtle differences among trees, ordered trees, and binary trees.

Topic 5.5

Walking binary trees

Preorder/Postorder walk over binary trees

We have more structure in binary trees. Let us write the algorithm for walks again.

Algorithm 5.3: PreOrderWalk(n)

```
if n == Null then
    return
visit(n);
PreOrderWalk(left(n));
PreOrderWalk(right(n));
```

Algorithm 5.4: PostOrderWalk(n)

```
if n == Null then
    return
PostOrderWalk(left(n));
PostOrderWalk(right(n));
visit(n);
```

Exercise 5.12

Are the above programs tail-recursive?

Inorder walk of binary trees

Definition 5.18

In an inorder walk of a binary tree, we visit the node after visiting the left subtree and before visiting the right subtree.

Algorithm 5.5: InOrderWalk(n)

```
if n == Null then
    return
InOrderWalk(left(n));
visit(n);
InOrderWalk(right(n));
```

Exercise 5.13

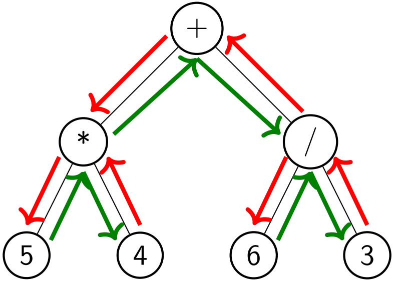
Given complete binary trees with 7 nodes, label the nodes such that the preorder, inorder, or postorder walks produce the sequence 1,2,...,7.

Application : Printing an expression

To print an expression (without unary minus), we need to visit the nodes in inorder.

Algorithm 5.6: PrintExpression(n)

```
1 if n is leaf then
2     print(label(n));
3     return
4 print("(");
5 PrintExpression(left(n));
6 print(label(n));
7 PrintExpression(right(n));
8 print(")");
```



((5 * 4) + (6 / 3))

Exercise 5.14

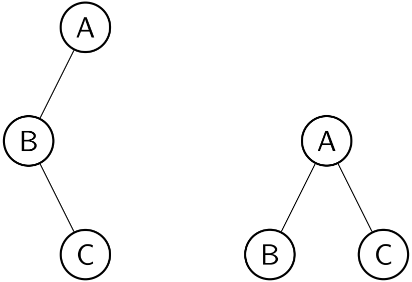
a. Modify the above algorithm to support unary minus.

b. What will happen if "**if**" at line 1 is replaced by "**if** $n == NULL$ **then return**"?

Commentary: The order of the walk is the pattern of recursive calls and actions on nodes. An application may need a mixed action pattern. In the above printing example, we need to print parentheses before and after making recursive calls. The parentheses are printed pre/post-order. All three walks are present in the above algorithm.

Many trees have the same walks

The following two ordered trees have the same preorder walks.



Commentary: Answer:

For postorder:

```
                A
               /
     A        C
    / \        \
   B   C        B
```

For inorder:

```
      A       B
     / \     / \
    B   C   D   A
   / \         / \
  D   E       E   C
```

For postorder and preorder:

```
  A   A
 /     \
C       C
 \     /
  B   B
```

Exercise 5.15

a. Give two binary trees that have the same postorder walks.

b. Give two binary trees that have the same inorder walks.

c. Give two binary trees that have the same postorder and preorder walks.

CS213/293 Data Structure and Algorithms 2025

Lecture 6: Binary search tree (BST)

Instructor: Ashutosh Gupta

IITB India

Compile date: 2025-09-13

Ordered dictionary

Recall: There are two kinds of dictionaries.

- Dictionaries with unordered keys
  - We use hash tables to store dictionaries for unordered keys.
- Dictionaries with ordered keys
  - Let us discuss the efficient implementations for them.

Recall: Dictionaries via ordered keys on arrays

- Searching is $O(log\ n)$
- Insertion and deletion is $O(n)$
  - Need to shift elements before insertion/after deletion

Can we do better?

Topic 6.1

Binary search trees

Binary search trees (BST)

Definition 6.1

A binary search tree is a binary tree T such that for each $n \in T$

- n is labeled with a key-value pair of some dictionary,
  - (if $label(n) = (k, v)$, we write $key(n) = k$)
- for each $n' \in descendants(\ left(n)\ )$, $key(n') \leq key(n)$, and
- for each $n' \in descendants(\ right(n)\ )$, $key(n') \geq key(n)$.

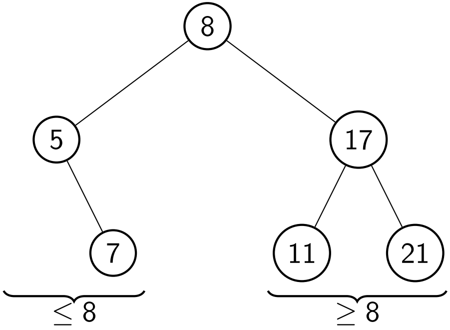
Note that we allow two entries to have the same keys. The same key can be in either of the subtrees.

Commentary: We assume $descendants(Null) = \emptyset$.

Example: BST

Example 6.1

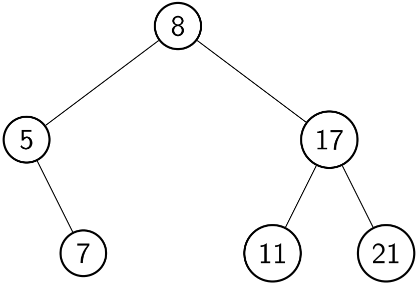
In the following BST, we show only keys stored at the node.

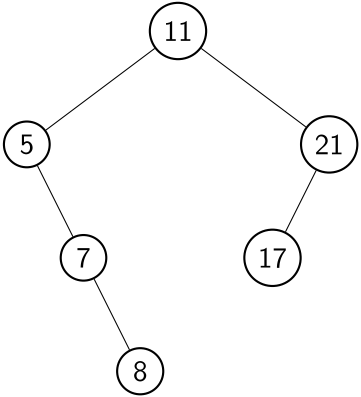


Example: many BSTs for the same data

Example 6.2

The same set of keys may result in different BSTs.

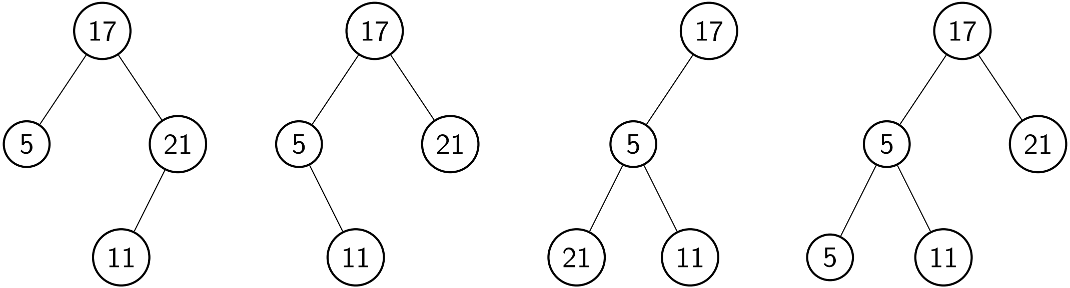




Exercise: Identify BST

Exercise 6.1

Which of the following are BSTs?



Topic 6.2

Algorithms for BST

Algorithms for BST

We need the following methods on BSTs

- search
- insert
- minimum/maximum
- successor/predecessor: Find the successor/predecessor key stored in the dictionary
- delete

Exercise 6.2

Give minimum and successor algorithms for sorted array-based implementation of a dictionary.

Commentary: Recall that we did not discuss algorithms for minimum and successor in our earlier discussion of unordered dictionaries, which are implemented using hash tables. Since we cannot define successor and minimum for unordered keys, the question of such algorithms does not arise. However, we do need them for operations on BSTs.

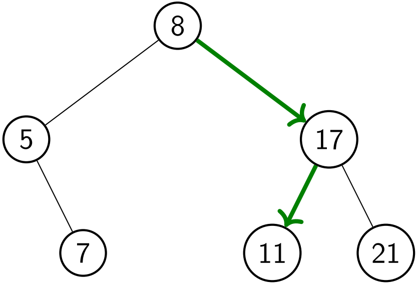
Searching in BST

Commentary: By the definition of BST, we are guaranteed that 11 will not occur in the left subtree of 8. This is the same reasoning as the binary search that we discussed earlier.

Example 6.3

Searching 11 in the following BST.

- We start at the root, which is node 8
- At node 8, go to the right child because $11 > 8$.
- At node 17, go to the left child because $11 < 17$.
- We find 11 at the node.

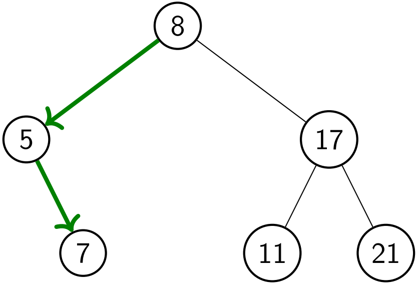


Unsuccessful search in BST

Example 6.4

Searching 6 in the following BST.

- We start at the root, which is node 8
- At node 8, go to the left child because $6 < 8$.
- At node 5, go to the right child because $6 > 5$.
- At node 7, go to the left child because $6 < 7$.
- Since node 7 has no left child the search fails.



Algorithm: Search in BST

Algorithm 6.1: SEARCH(BST T, int k)

```
n := root(T);
while n ≠ Null do
    if key(n) = k then
        break
    if key(n) > k then
        n := left(n)
    else
        n := right(n)
return n
```

- Running time is $O(h)$, where h is height of BST.
- If there are n keys in the BST, the worst case running time is $O(n)$.

Commentary: Answer:
a. We search in the BST. If the key is found on a node, then we start two(Why?) searches in both the subtrees of the found node. We recursively start the searches.

b. Find N in the following BST

```
1
 \
  2
   \
    ..
     \
      N
```

Exercise 6.3

a. Modify the above algorithm to find all occurrences of key k.

b. Give an input of SEARCH that exhibits worst-case running time.

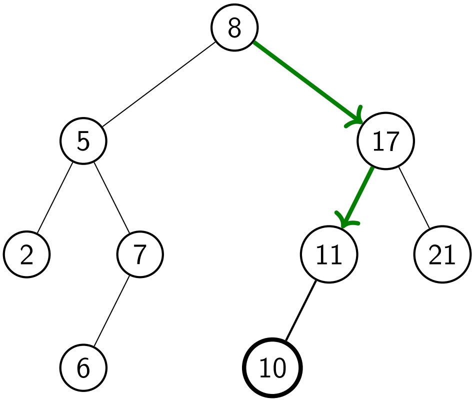
Topic 6.3

Insert in BST

Example: Insert in BST

Example 6.5

key 10 was not there in the BST and we want to insert 10. Where do we insert 10?



We always insert the new key as leaf.

Algorithm: Insert in BST

Algorithm 6.2: INSERT(BST T, Node n)

```
x := root(T); y := Null;
while  x ≠ Null do
    y := x;
    if  key(x) > key(n) then
        x := left(x)
    else
        x := right(x)
if y = Null then
    root(T) = n;
if key(y) > key(n) then
    left(y) := n
else
    right(y) := n
parent(n) = y
```

Exercise 6.4

a. What is the running time of the algorithm?

b. Give an order of insertion for the maximum tree height.

c. Give an order of insertion for the minimum tree height.

d. What does happen if key(n) already exists?

Commentary: Answer:
a. the same as search,
b. 1,2,3,4,5,...,n
c. $n/2, n/4, 3n/4, n/8, 3n/8, 5n/8, 7n/8, ..$
d. This algorithm always goes right. It is correct but may not be a good idea. It should randomly choose left or right.

Topic 6.4

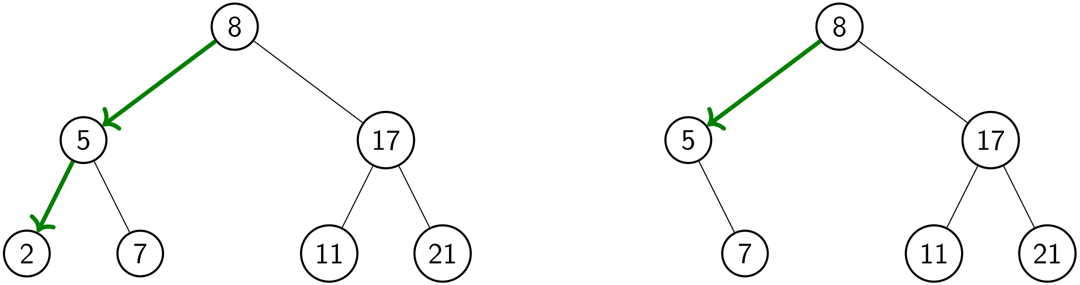
Minimum in BST

Example: minimum in BST

Commentary: Always go left to find a smaller node. As soon as we do not have a left child, we have found the minimum node.

Example 6.6

What is the minimum of the following BSTs?



Algorithm: Minimum in BST

The following algorithm computes the minimum in the subtree rooted at node n.

Algorithm 6.3: MINIMUM(Node n)

```
while n ≠ Null and left(n) ≠ Null do
    n := left(n)
return n
```

- Runtime analysis is the same as SEARCH.

Exercise 6.5

Modify the above algorithm to compute the maximum

Commentary: Note that $key(n') \leq key(n) \leq key(n'')$ where $n'' \in descendants(right(n))$

Correctness of MINIMUM

Theorem 6.1

If $n \neq Null$, the returned node by MINIMUM(n) has the minimum key in the subtree rooted at n.

Proof.

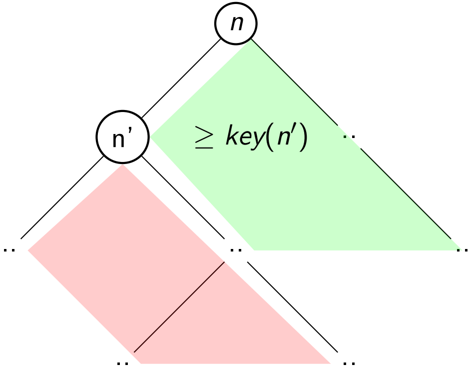
If $left(n) = Null$, $key(n)$ is the minimum key.

Otherwise, we go to $n' = left(n)$. Any node not in $descendants(n')$ must have a larger key than $key(n')$.(Why?)

So the minimum of $descendants(n')$ is the overall minimum.

This argument continues to hold for any number of iterations of the loop. (induction)

Therefore, our algorithm will compute the minimum.



Topic 6.5

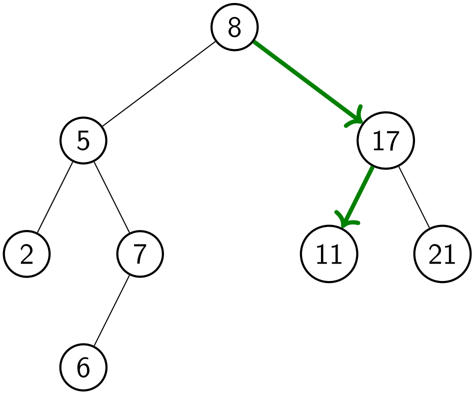
Successor in BST

Example: successor in BST

We now consider the problem of finding the node that has the successor key of a given node.

Example 6.7

Where is the successor of 8?

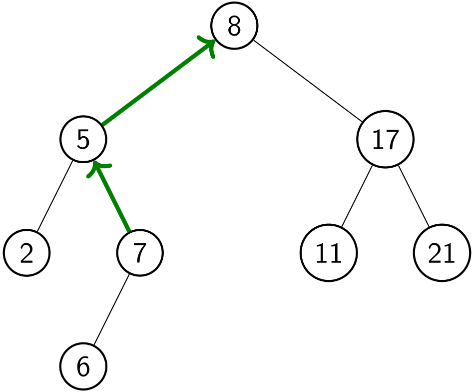


Observation: Minimum of right subtree.

Example: successor in BST(2)

Example 6.8

Where is the successor of 7?



Exercise 6.6

a. When do we not have the successor in the right subtree?

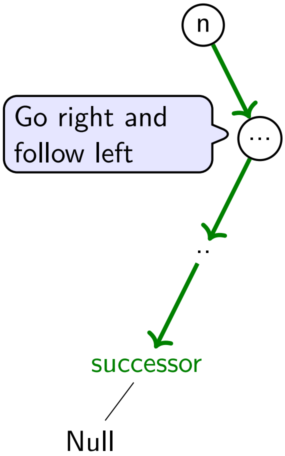
b. If the successor is not in the right subtree, where else can it be?

Commentary: Successor may be found in two possible areas. The second case is slightly difficult to understand, where the successor is one of the ancestors. It is the closest ancestor that is bigger than *n*. This happens when the path turns right first time. The formal proof is at the end of the slides.

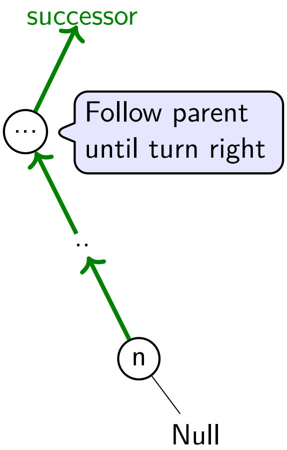
Cases for the location of the successor

Finding a successor to *n*

Case 1: If there is a right subtree:



Case 2: If there is no right subtree:



Successor in BST

Algorithm 6.4: SUCCESSOR(BST T, node n)

if $right(n) \neq Null$ **then**
  return MINIMUM($right(n)$)
while $parent(n) \neq Null$ *and* $right(parent(n)) = n$ **do**
  $n := parent(n)$;
return $parent(n)$

Exercise 6.7

a. Modify the above algorithm to compute predecessor
b. What is the running time complexity of SUCCESSOR?
c. What happens when we do not have any successor?
d. What is returned if multiple keys have the same value?
e. What is the connection between the above algorithm and in-order walk?
f. Can we modify the above algorithm to find the strict successor?

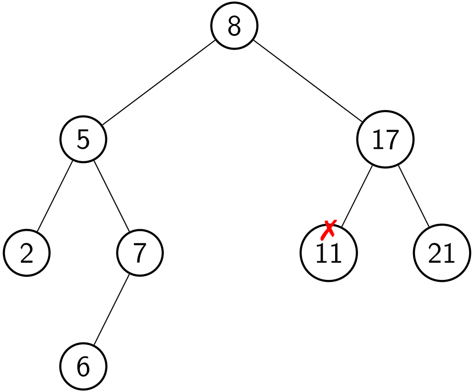
Topic 6.6

Deletion

Example: deleting a leaf

Example 6.9

How can we delete leaf 11?

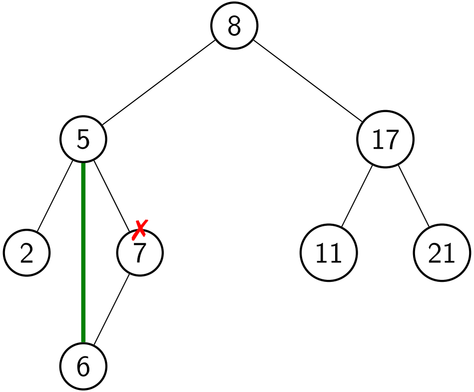


We delete leaf 11 by simply removing the node.

Example: deleting a node with a single child

Example 6.10

How can we delete node 7, which has a single child?

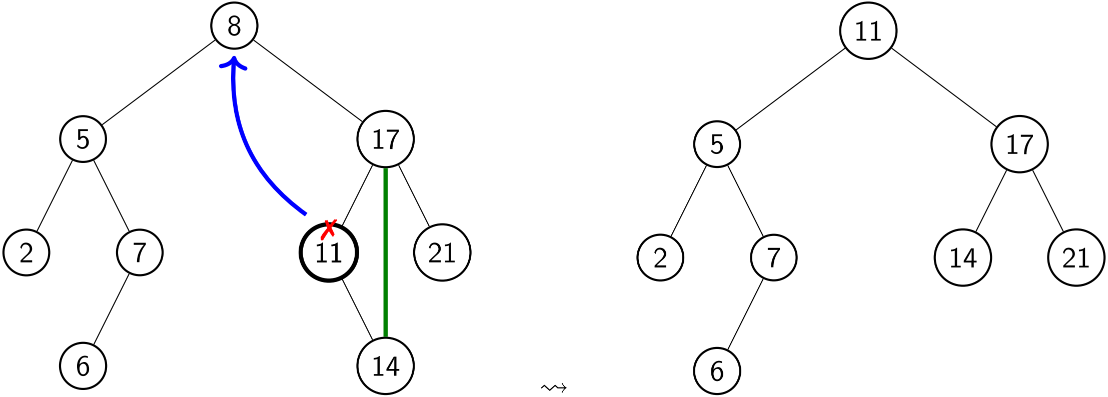


We delete node 7 by making 6 the child of 5 and removing the node.

Example: deleting a node with both children

Example 6.11

How can we delete node 8, which has both the children?



We delete node 8 by removing 11, which is the successor of 8, and moving the data of 11 to 8.

Algorithm: delete in BST*

Algorithm 6.5: DELETE(BST T, Node n)

$y := (left(n) = Null \vee right(n) = Null)\ ?\ n\ :\ \textsc{Successor}(T, n);$ // y will be deleted
if $y \neq n$ **then**
  $key(n) := key(y)$ // copy all data on y
$x := (left(y) = Null)\ ?\ right(y)\ :\ left(y);$ //x is the child of y or x is Null
if $x \neq Null$ **then**
  $parent(x) = parent(y)$ //y is not a leaf, update the parent of x
if $parent(y) = Null$ **then**
  $root(T) = x$ // y was the root, therefore x is root now
else
  if $left(parent(y)) = y$ **then**
    $left(parent(y)) := x$ //Remove y from the tree
  else
    $right(parent(y)) := x$ //Remove y from the tree

Exercise 6.8

a. How can we delete by key instead of node? Does it change the complexity? b. Do we need to free y?

CS213/293 Data Structure and Algorithms 2025

Lecture 8: Heap

Instructor: Ashutosh Gupta

IITB India

Compile date: 2025-09-13

Topic 8.1

Priority queue

Scheduling problem

On a computational server, users are submitting jobs to run on a single CPU.

- A user also declares the expected run time of the job.
- Jobs can be preempted.

Policy: shortest remaining processing time, which allows interruption of a job if a new job with a smaller run time is submitted.

The policy minimizes average waiting time.

Scheduling problem operations

We need the following operations in the scheduling problem.

- Update the remaining time in every tick
- Delete a job when the remaining time is zero
- Find the next job to run
- Insert a job when it arrives

Definition 8.1

In a priority queue, we dequeue the highest priority element from the enqueue elements with priorities.

Interface of priority queue

https://en.cppreference.com/w/cpp/container/priority_queue

- `priority_queue<T,Container,Compare> q` : allocates new queue q
- `q.push(e)` : adds the given element e to the queue.
- `q.pop()` : removes the highest priority element from the queue.
- `q.top()` : access the highest priority element.

- `Container` class defines the physical data structure where the queue will be stored. The default value is `Vector`.
- `Compare` class defines the method of comparing priorities of two elements.

Topic 8.2

Implementations of priority queue

Implementation using unsorted linked list/array

In case we use a linked list,

- We implement `q.push` by inserting the element at the front of the linked list, which is $O(1)$ operation.
- We need to scan the entire list to find the maximum for implementing `q.pop` and `q.top`

Exercise 8.1

How will we implement a priority queue over unsorted arrays?

Implementation using sorted linked list/array

In case we use a linked list,

- The maximum will be at the end of the list. We can implement `q.pop` and `q.top` in $O(1)$.
- However, `q.push(e)` needs to scan the entire list to find the right place to insert e, which is $O(n)$ operation.

Priority queue

The priority queue is one of the fundamental containers.

Many other algorithms assume access to efficient priority queues.

We will define a data structure heap that provides an efficient implementation for the priority queue.

Commentary: The heap is like the red-black tree, which provides an efficient implementation for ordered maps.

Topic 8.3

Heap - partial sorting!

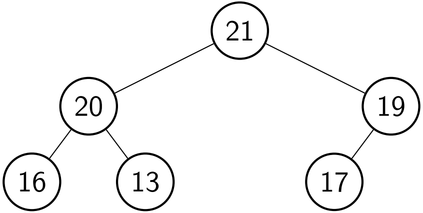
Heap

Definition 8.2

A heap T is a binary tree such that the following holds.

- (structural property) All levels are full except the last one and the last level is left filled.
- (heap property) for each non-root node n, $key(n) \leq key(parent(n))$.

Example 8.1



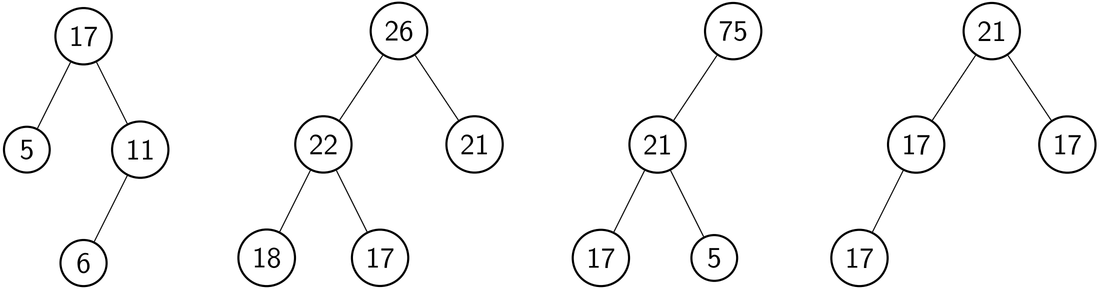
Exercise 8.2

a. Show that nodes on a path from the root to a leaf have keys in non-increasing order.

b. The above definition is called maxheap. Can we symmetrically define minheap?

Exercise: identify heap

Exercise 8.3
Which of the following are Heaps?



Algorithm: maximum

Algorithm 8.1: $\textsc{Maximum}$(Heap T)

return $T[0]$

- Correctness
  - Let us suppose the maximum is not at the root.
  - There is a node n that has maximum key but $parent(n)$ has a smaller key, which violates heap condition.
  - Contradiction.
- Running time is $O(1)$.

Height of heap

Let us suppose a heap has n nodes and height h.

The number of nodes in a complete binary tree of height h is $2^{h+1} - 1$.

Therefore,

$$2^h - 1 < n \leq 2^{h+1} - 1.$$

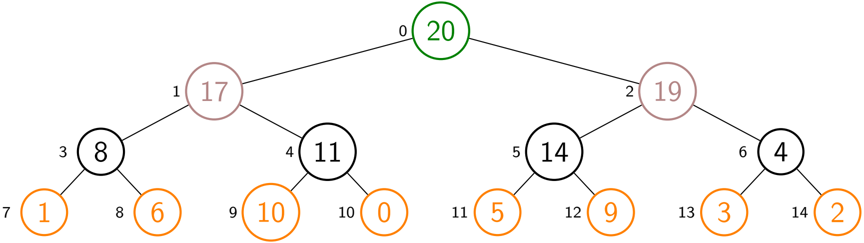
Therefore $h = \lfloor \log_2 n \rfloor$

Exercise 8.4

Give an example of a heap that touches the lower bound.

Storing heap

Let us number the nodes of a heap in the order of level.



parent$(i) = (i-1)/2$, *left*$(i) = 2i+1$, and *right*$(i) = 2i+2$.

We place the nodes on an array and traverse the heap using the above equations.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
20	17	19	8	11	14	4	1	6	10	0	5	9	3	2

Since the last level is left filled, we are guaranteed the nodes are contiguously placed.

Instead of writing *key*(i) for node i in heap T, we will write $T[i]$ to indicate the key.

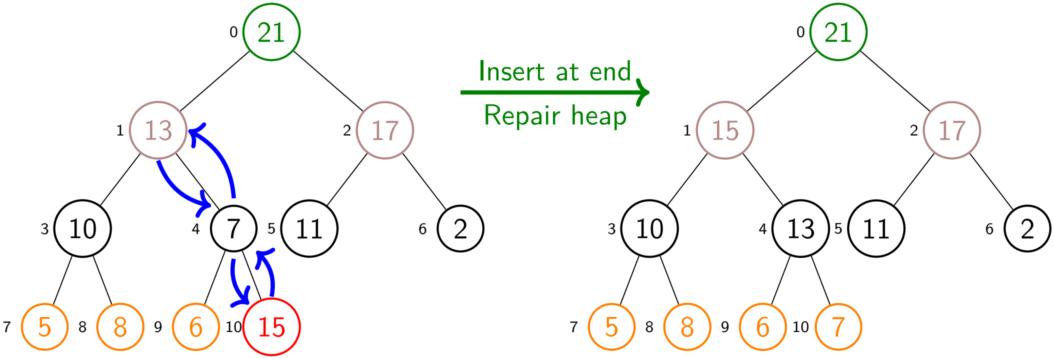
Topic 8.4

Insert in heap – jostling to front

Example: insert in heap

Example 8.2

Where do we insert 15?



- Insert at the first available place, which is easy to spot. (Why?)
- Move up the new key if the heap property is violated.

Commentary: At the time of insert, we may have not have enough space in T. On insert we may need to apply expand operations like stack and queue.

Algorithm: Insert

Algorithm 8.2: INSERT(Heap T, key k)

$i := T.size$;
$T[i] := k$;
while $i > 0$ *and* $T[parent(i)] < T[i]$ **do**
    SWAP(T, parent(i), i);
    $i := parent(i)$
$T.size := T.size + 1$;

- Correctness
  - Structural property holds due to the insertion position.
  - Due to the heap property of input T, the path to i (not including i) the nodes must be in non-increasing order.
  - Let i_0 be the value of i when the loop exits.
  - INSERT replaces the keys of the nodes in the path from i_0 to $T.size$ with the keys of their parents, which implies the keys do not decrease at the internal nodes.
  - Therefore, no introduction of a violation.
  - Therefore, we will have a heap at the end.
- Running time is $O(\log T.size)$.

Exercise 8.5

Why do we need the phrase “not including” and “internal” in the above proof?

Topic 8.5

Heapify: fix the almost heaps

Heapify : a basic operation on a heap

Input to HEAPIFY:

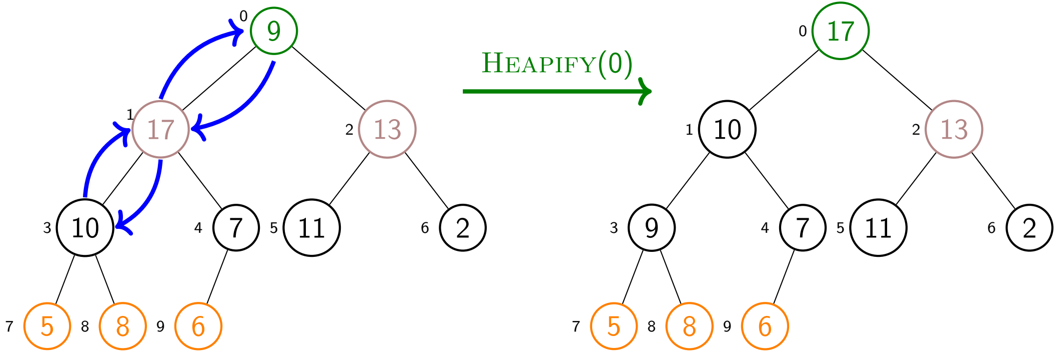
- Let i be a node of a binary tree T with the structural property of heap
- Let us suppose the binary trees rooted at $left(i)$ and $right(i)$ are valid heaps.
- $T[i]$ may be smaller than its children and violates the heap property.

Output of HEAPIFY:
HEAPIFY makes the binary tree rooted at i a heap by pushing down $T[i]$ in the tree.

Example: HEAPIFY

Example 8.3

The trees rooted at positions 1 and 2 are heaps. We have a violation at position 0. Heapify will fix the problem by moving the key down.



- Keep moving down to the child which has the maximum key. (Why?)

Algorithm: Heapify

Algorithm 8.3: HEAPIFY(Heap T, i)

$c := \textsc{IndexWithLargestKey}(T, i, left(i), right(i))$ //assume $T[i] = -\infty$ if $i \geq T.size$.
if $c == i$ **then** **return**;
$\textsc{Swap}(T, c, i)$;
$\textsc{Heapify}(T, c)$;

- Correctness
  - Same as insert, but we are pushing down.
- Running time is $O(\log T.size)$.

Commentary: Assumption $T[i] = -\infty$ if $i \geq T.size$ is a convenience of notation. We may have a situation, where the $T[i]$ exists and has some key. Without loss of correctness, we can interpret them as if the key is $-\infty$. We will need this interpretation later for HEAPSORT.

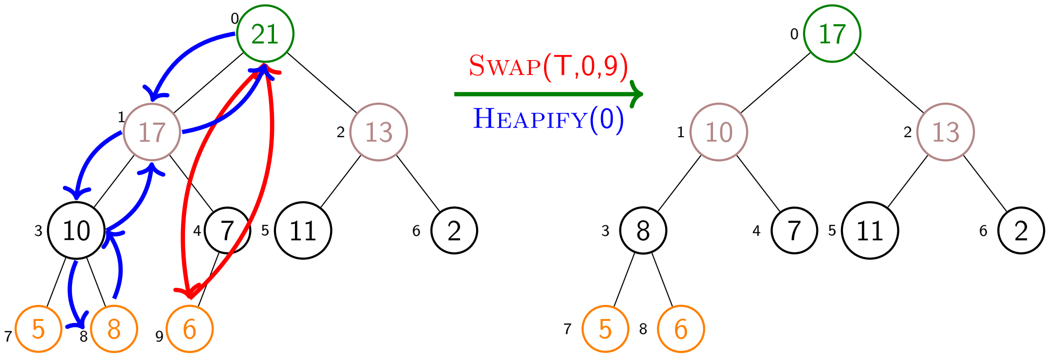
Topic 8.6

Delete maximum in heap

Example: DELETEMAX

Example 8.4

Let us delete 21 at position 0.



- Swap with the last position, delete the last position, and run HEAPIFY.

Algorithm: DeleteMax

Algorithm 8.4: $\textsc{DeleteMax}$(Heap T)

```
Swap(T, 0, T.size − 1);
T.size := T.size − 1;
Heapify(T, 0);
return T[T.size];
```

- Correctness
  - The maximum element is removed and heapify returns a heap.
- Running time is $O(\log T.size)$.

Topic 8.7

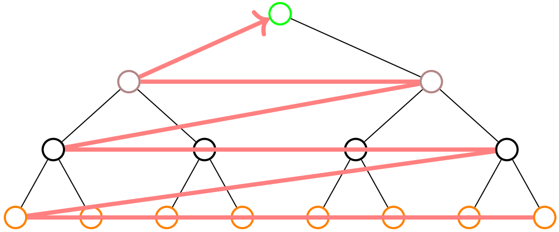
Build heap

Build heap https://en.cppreference.com/w/cpp/algorithm/make_heap

- Input: A binary tree T that has the structural property
  - If the structural property holds, then the T is an array
- Output: A heap over elements of T

Algorithm: BuildHeap

Order of processing in BuildHeap.



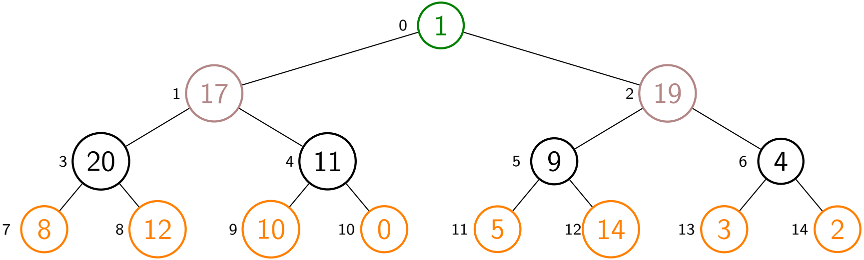
Algorithm 8.5: BuildHeap(Heap T)

for $i := T.size - 1$ **down to** *0* **do**
  Heapify(T, i)

Example: BUILDHEAP

Example 8.5

Consider sequence 1 17 19 20 11 9 4 8 12 10 0 5 14 3 2. Let us fill them in the following tree.



BUILDHEAP traverses the tree bottom up. HEAPIFY calls execute only the following swaps.

- HEAPIFY(T,5): SWAP(T,5,12)
- HEAPIFY(T,1): SWAP(T,1,3)
- HEAPIFY(T,0): SWAP(T,0,1); SWAP(T,1,3); SWAP(T,3,8);

The other calls to HEAPIFY will not apply any swaps.

Correctness of BuildHeap

- We do not change the structure of T in BuildHeap, therefore the tree at any i has the structural property.
- Correctness by induction
  - **Base case:**
    If i does not have children, it is already a heap.
  - **Induction step:**
    We know $left(i) > i$ or $right(i) > i$.
    Due to the induction hypothesis, both the subtrees are heap before processing i.
    The tree at i has structural property. Therefore, Heapify(T, i) will return a heap rooted at i.

Running time of BuildHeap

Let us suppose T is a complete tree with n nodes.

Recall: Heapify for a node at height h has $O(h)$ swaps.

At height h the number of nodes is $\lceil n/2^{h+1} \rceil$ and the height of T is $\lfloor \log n \rfloor$.

The total running time of BuildHeap is

$$\sum_{h=0}^{\lfloor \log n \rfloor} O(h) \lceil n/2^{h+1} \rceil = O\left(\frac{n}{2} \sum^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

Commentary: We used identities $O(f)g = O(fg)$ and $O(f) + O(g) = O(f+g)$.

Since $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$, the running time is $O(n)$.

Calculation to show $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$

We know

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}$$

After differentiating over x,

$$\sum_{h=0}^{\infty} hx^{h-1} = \frac{1}{(1-x)^2}$$

After multiplying with x,

$$\sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2}$$

After putting $x = 1/2$,

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

Topic 8.8

Heapsort

HEAPSORT

Algorithm 8.6: HEAPSORT(Tree T)

$T.size = |\text{nodes of T}|$;
BUILDHEAP(T);
while $T.size > 0$ **do**
  DELETEMAX(T)

- Since DELETEMAX moves maximum to $T.size - 1$ position, the array is sorted in place.
- Running time:
  - BUILDHEAP is $O(n)$
  - DELETEMAX(T) is $O(\log i)$ at size i.
- Total running time: $O(n \log n)$.

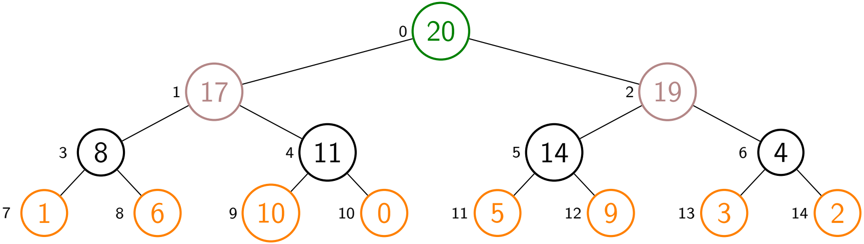
Exercise 8.6

Both BUILDHEAP and the above loop have iterative runs of HEAPIFY.
Why are their running time complexities different?

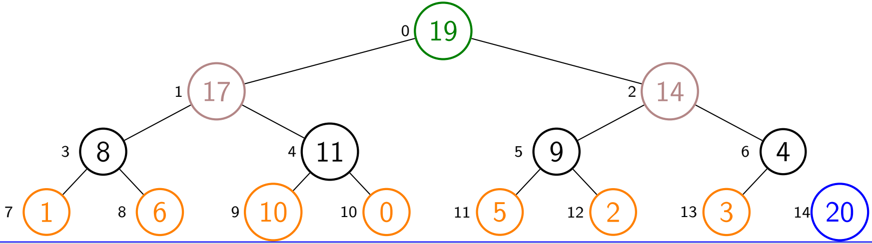
Commentary: Please solve the above exercise to clearly understand the relevant mathematics.

Example: HEAPSORT

Consider the following Heap obtained after running BUILDHEAP.

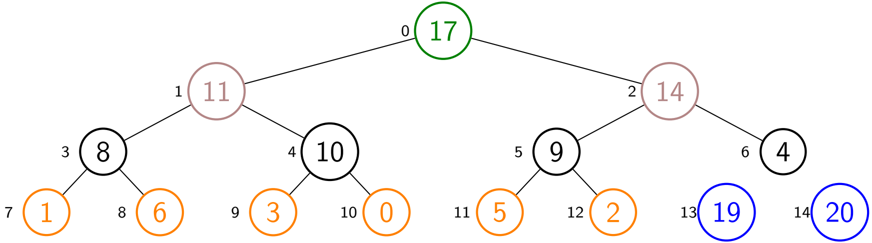


After the first DELETEMAX,



Example: Heapsort(2)

After the second DeleteMax,



DeleteMax has placed 19 and 20 at their sorted position.