CS 234: Data Types and Structures

Naomi Nishimura

Module 8

Date of this version: November 11, 2019

WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.

Case study

Problem: Collect bids for a construction project, selecting the lowest bid.

Recipe for user/plan

1. Determine types of data and operations.
2. For each type, choose/modify/create an ADT.
3. Develop pseudocode algorithm using ADT operations.
4. Calculate cost of algorithm with respect to costs of operations.
5. Using information from provider, choose best option.

ADT Priority Queue

Data: (key, element) pairs where

- keys are orderable but not necessarily distinct, and
- elements are any data.

Preconditions: For all *P* is a priority queue, *Key* is a key, and *Element* is an element; for *LOOK_UP_MIN* and *DELETE_MIN*, *P* is not empty.

Postconditions: Mutation by *ADD* (adds pair with key *Key* and element *Element*) and *DELETE_MIN* (deletes a pair with minimum key).

Name	Returns
CREATE()	a new empty priority queue
IS_EMPTY(P)	*True* if empty, else *False*
LOOK_UP_MIN(P)	pair with minimum key
ADD(P, Key, Element)	
DELETE_MIN(P)	pair with minimum key

Contiguous implementations of priority queues

Contiguous implementation 1

Data structures:

- Array with all data items stored contiguously, starting at 0
- Variable *First* storing index of first empty slot

Contiguous implementation 2

Data structures:

- Array with all data items stored contiguously, starting at 0
- Variable *First* storing index of first empty slot
- Variable *Min* storing index of an item with minimum priority

Contiguous implementation 3

Data structures:

- Array with all data items stored contiguously, starting at 0, in decreasing order by key
- Variable *First* storing index of first empty slot

Linked implementations of priority queues

Linked implementation 1

Data structures:

- Variable *Head* storing a pointer to a linked list of nodes, each storing a data item and a pointer to the next node in the list

Linked implementation 2

Data structures:

- Variable *Head* storing a pointer to a linked list of nodes, each storing a data item and a pointer to the next node in the list
- Variable *Min* storing a pointer to a node storing a data item with minimum priority

Linked implementation 3

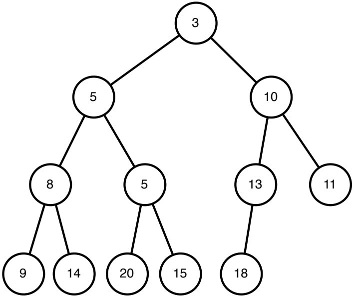
Data structures:

- Variable *Head* storing a pointer to a linked list of nodes, each storing a data item and a pointer to the next node in the list, in increasing order by key

Data structure: Heap

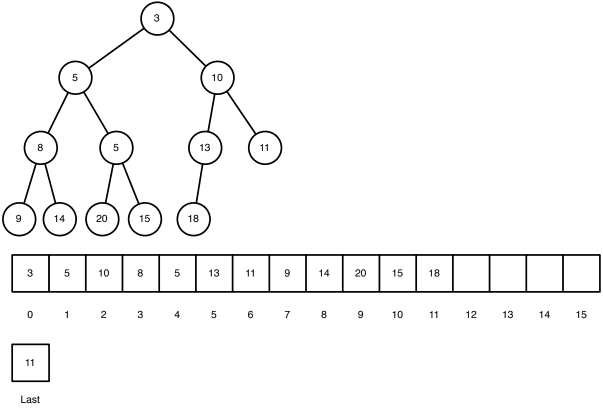
A binary tree satisfies the **heap-order property** if for each node, the value stored at the node is no greater than that stored in either child.

A **heap** is a complete binary tree that satisfies the heap-order property.



Using a heap to implement the ADT Priority Queue: Store a (key, element) pair at each node, ensuring that the keys satisfy the heap-order property.

Implementing a heap using an array implementation of ADT Binary Tree



Note: Ilustrations show keys only, not elements.

Implementing ADT Priority Queue using a heap

LOOK_UP_MIN(P)

- Return the data item in the root of tree.
- $\Theta(1)$ to find item in position 0 in the array.

ADD(P, Key, Element)

- To preserve completeness, add at next leaf position.
- $\Theta(1)$ to calculate position ($Last + 1$) and update $Last$
- Problem: Heap-order property violated.

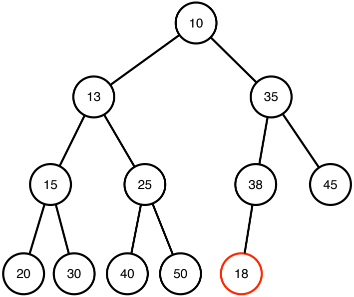
DELETE_MIN(P)

- To preserve heap-order property, remove the root.
- Problem: What remains is not a tree.

Implementing *ADD(P, Key, Element)* in a heap

"Bubble up", fixing heap-order property on path from leaf to root by swapping the values stored in a node and its parent.

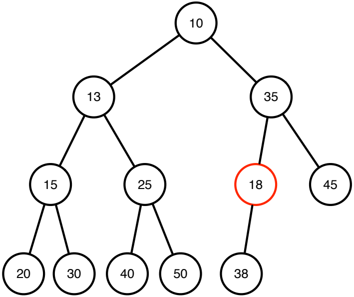
Here 18 has just been added.



Implementing *ADD(P, Key, Element)* in a heap

"Bubble up", fixing heap-order property on path from leaf to root by swapping the values stored in a node and its parent.

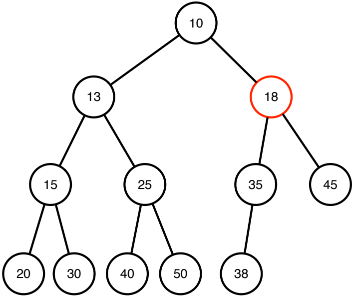
Here 18 has just been added.



Implementing *Add(P, Key, Element)* in a heap

"Bubble up", fixing heap-order property on path from leaf to root by swapping the values stored in a node and its parent.

Here 18 has just been added.



Augmenting the ADT Binary Tree

We will augment the ADT Binary Tree in order to support:

- Storing of both keys and elements at each node.
- Access to either key or element.

Note: Only keys are ordered using the heap order property.

Modified ADT Binary Tree operations:

- *ADD_LEAF(B, Par, Side, Key, Element)*
- *SET_VALUES(B, Node, Key, Element)*

New operations:

- *KEY(B, Node)*
- *ELEMENT(B, Node)*
- *SIDE(B, Node)* - produces *Left* or *Right*
- *SWAP_NODE_VALUES(B, One, Two)* - exchanges both key and elements in nodes *One* and *Two*

Heap-specific operations

Each of the operations returns a location of a node (hence an array index):

- *LAST_LEAF(B)* produces the location of the last leaf
- *PREVIOUS_LEAF(B)* produces the location of what will be the last leaf if the last leaf is removed
- *NEXT_LEAF(B)* produces the location of the last leaf if a new leaf is added

These operations make use of the fact that a heap is always a complete binary tree.

Constant-time implementations in a heap

We use a variable *Last* to store the index of the array position of the last leaf in the tree.

- *LAST_LEAF(B)* returns *Last*
- *PREVIOUS_LEAF(B)* returns *Last* - 1
- *NEXT_LEAF(B)* returns *Last* + 1

Recall constant-time operations from array implementation of a binary tree: *ROOT*, *PARENT*, *LEFT_CHILD*, *RIGHT_CHILD*

New operations will also all be constant-time operations.

Pseudocode for *Add(P, Key, Element)*

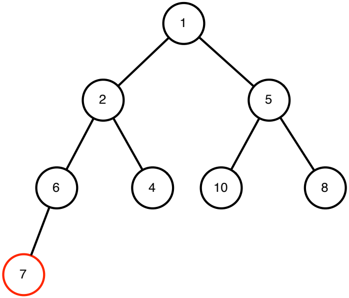
Last $\leftarrow$ *Next_Leaf(B)*
Add_Leaf(B, Parent(B, Last), Side(B, Last), Key, Element)
Curr $\leftarrow$ *Last*
Par $\leftarrow$ *Parent(B, Curr)*
while *Par* $\neq$ *False* ***and*** *Key(B, Curr)* $<$ *Key(B, Par)*
    Swap_Node_Values(B, Curr, Par)
    Curr $\leftarrow$ *Par*
    Par $\leftarrow$ *Parent(B, Curr)*

Implementing *DELETE_MIN(P)* in a heap

Delete value in root, move value in last leaf to root.

"Bubble down", fixing heap-order property on path from root to leaf using *SWAP_NODE_VALUES*.

Be careful to check both children of a node.

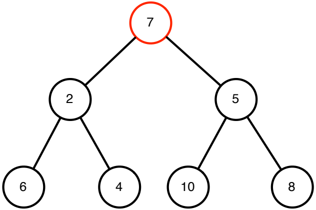


Implementing *DELETE_MIN(P)* in a heap

Delete value in root, move value in last leaf to root.

"Bubble down", fixing heap-order property on path from root to leaf using *SWAP_NODE_VALUES*.

Be careful to check both children of a node.

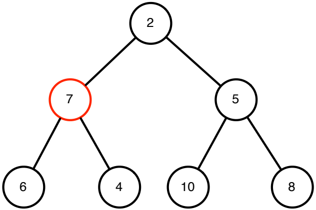


Implementing *Delete_Min(P)* in a heap

Delete value in root, move value in last leaf to root.

"Bubble down", fixing heap-order property on path from root to leaf using *Swap_Node_Values*.

Be careful to check both children of a node.

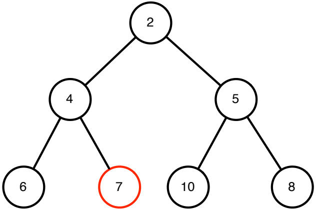


Implementing *DELETE_MIN(P)* in a heap

Delete value in root, move value in last leaf to root.

"Bubble down", fixing heap-order property on path from root to leaf using *SWAP_NODE_VALUES*.

Be careful to check both children of a node.



Delete_Min algorithm

Ideas used in the pseudocode (to follow):

- Swap the values in the last leaf and the root.
- Delete the last leaf and update the index for *Last*.
- Starting at the root as the current node, compare the values of the current node and its children.
- If the current node does not have the smallest value, swap its value with the smaller value held in a child and make the child the current node.
- If the current node has only a left child and not a right child, swap its value with the child's value if the child's value is smaller.
- Return the minimum value.

DELETE_MIN pseudocode, part 1

$Min\_Key \leftarrow \textsc{Key}(B, \textsc{Root}(B))$
$Min\_Element \leftarrow \textsc{Element}(\textsc{Root}(B))$
$\textsc{Swap\_Node\_Values}(B, Root(B), Last)$
$\textsc{Delete\_Node}(B, Last)$
$Last \leftarrow \textsc{Previous\_Leaf}(B)$
$Curr \leftarrow Root(B)$
$Left \leftarrow \textsc{Left\_Child}(B, Curr)$
$Right \leftarrow \textsc{Right\_Child}(B, Curr)$
$Key \leftarrow \textsc{Key}(B, Curr)$
$Stop \leftarrow False$

DELETE_MIN pseudocode, part 2

```
while Left ≠ False and not Stop
    Min_Child ← Left
    if Right ≠ False and KEY(B, Right) < KEY(B, Left)
        Min_Child ← Right
    if KEY(B, Min_Child) < Key
        SWAP_NODE_VALUES(B, Curr, Min_Child)
        Curr ← Min_Child
        Left ← LEFT_CHILD(B, Curr)
        Right ← RIGHT_CHILD(B, Curr)
    else
        Stop ← True
return Min_Key, Min_Element
```

Extracting pairs ordered by key

Goal: Produce the pairs in the priority queue in sorted order by key.

Options for producing sorted pairs:

- Write an algorithm using existing ADT operations.
- Augment the ADT by adding a new operation.

Using existing ADT operations:

- Repeatedly use *ADD* until all values have been entered.
- Repeatedly use *DELETE_MIN*.

Analysis:

- $\Theta(n \log n)$ for n *ADD* operations
- $\Theta(n \log n)$ for n *DELETE_MIN* operations

Provider view

Can we find a faster way to make a heap out of n elements?

Augmenting ADT Priority Queue

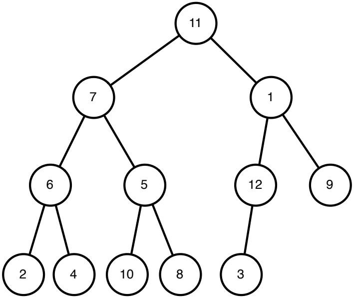
Form heap out of a bunch of (key, element) pairs.

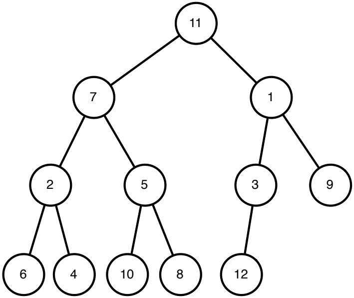
The *HEAPIFY* operation forms a heap out of an array of items.

Idea:

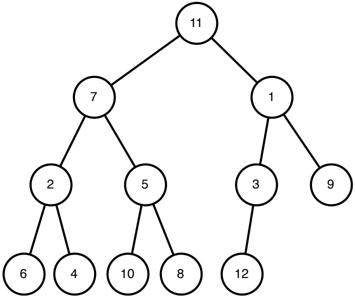
- Place all items into the structure.
- Fix heap-order property from bottom up.
- Observe that leaves are heaps of height 0.
- At phase i, form heaps of height at most i from two heaps of height at most $i-1$.

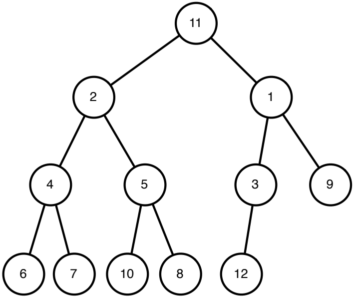
HEAPIFY example, phase 1



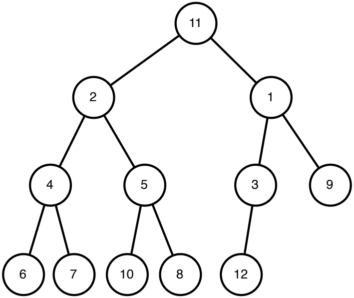


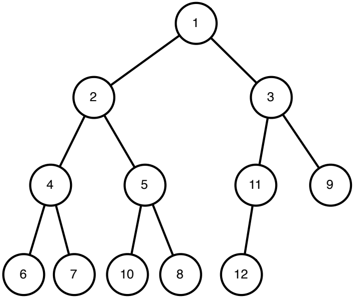
HEAPIFY example, phase 2





Heapify example, phase 3





Linear-time *Heapify* analysis

- Placement into structure takes $\Theta(n)$ time total
- Logarithmic number of phases, in phase i forming at most $n/2^{i+1}$ heaps of height i each
- Cost of making one heap of height i is in $\Theta(i)$, bounded above by some ci
- Total cost of phases is at most $\sum_{i=1}^{\lfloor \log n \rfloor} ci \cdot \frac{n}{2^{i+1}} \leq cn(\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \cdots)$, which is linear in n
- Total cost in $O(n)$

Sorting using a heap

Heapsort:

- Build a heap using *Heapify*.
- Extract pairs in order using *Delete_Min* repeatedly.

Analysis on n elements:

- $O(n)$ time
- n iterations, each $O(\log n)$ time

Total cost: $O(n \log n)$ time

Module summary

Topics covered:

- Case study: Bids
- ADT Priority Queue
- Contiguous implementations
- Linked implementations
- Data structure: Heap
- Extracting pairs ordered by key
- Sorting using a heap