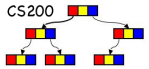
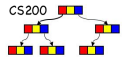


CS200: Tables and Priority Queues

Prichard Ch. 12

Value Oriented Data Structures



- Value-oriented operations are very common:
  - Find John Smith's facebook
  - Retrieve the student transcript of Ruth Mui
  - Register Jack Smith for an Amazon Cloud account
  - Add a user to a database (e.g., Netflix database).
- To support such uses: Arrange the data to facilitate search/insertion/deletion of an item given its search key

Example: Table of Student Points

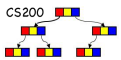
Student ID	Student First Name	Student Last Name	Exam 1	Exam 2
001245	Jake	Glen	67	89
001247	Parastoo	Mahgreb	87	78
001256	Wayne	Dwyer	90	96
012345	Bob	Harding	45	50
022356	Mary	Laing	95	97

- Each row is a record
- Each column is a field
- Student ID is the search key field

Search Keys

- In many applications the search key must be unique to a *single* record
- Records should be organized to facilitate search for an item by search key
- The search key of a record must not change while it is in the table. Why?

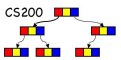
Table Records




```
public abstract class KeyedItem<KT extends
   Comparable <? super KT>>
//KT is constrained to be a type that implements comparable or is a
//subclass of a type which does so
{
   private KT searchKey;

   public KeyedItem(KT key) {
       searchKey = key; }  //constructor

   public KT getKey() {
       return searchKey; }  //accessor
}
```

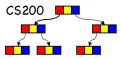
There is no method to set the search key. Why?

Table Record Example



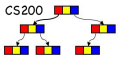

```
public class User extends KeyedItem<String> {
   private String StudentID;     // search key
   private String firstName;
   private String lastName; …

   public User(String userID, String _firstName, …)  {
       super(userID);//Why is super used here?
       firstName = _firstName; …
   }//constructor
```


Table Interface



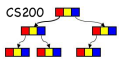

```
public interface TableInterface<T extends KeyedItem<KT>,
          KT extends Comparable <? super KT>> {
   // Precondition for all operations:
   // No two items of the table have the same search key.
   // The table's items are sorted by search key
    // (actually not required)

   public boolean tableIsEmpty();
   // Determines whether a table is empty.
   // Postcondition: Returns true if the table is empty;
   // false otherwise

   public int tableLength();
   // Determines the length of a table.
   // Postcondition: Returns number of items in the table.
```


Table Interface (cont.)




```
public void tableInsert(T newItem)  throws
 TableException;
  // Inserts a record into a table in its proper sorted
  // order according to the item's search key.
  // Precondition: The record's (newItem's) search key
  // must be unique in the table.
  // Postcondition: If successful, newItem is in its
  // proper order in table.
  // Otherwise, throw TableException.

public boolean tableDelete(KT searchKey);
  // Deletes a record with search key KT from table.
  // Precondition: searchKey is the search key of item
  // Postcondition: If there is a record with KT in the
  // table, the item was deleted and method returns
  // true. Otherwise, table is unchanged; return false.
```

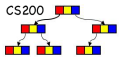
Table Interface (cont.)




```
public KeyedItem tableRetrieve(KT searchKey);
   // Retrieves a record with a search key KT from table.
   // Precondition: searchKey is search key for record to
   // be retrieved.
   // Postcondition: If the retrieval was successful,
   // table record with search key matching KT is returned.
   // If no such record exists, return null.

}  // end TableInterface
```

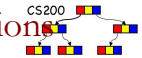

Reusing Data Structures

Possible Implementations



- Array sorted by search key
- Linked List sorted by search key
- Binary search tree

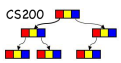
Performance of Table Implementations



	Search	Add	Remove
Sorted array-based	O(log n)	O(n)	O(n)
Unsorted array-based	O(n)	O(1)	O(n)
BST*	O(log n) [O(n)]	O(log n) [O(n)]	O(log n) [O(n)]

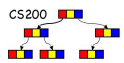
** Worst case behavior in []*

Priority Queues



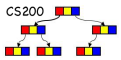
- Characteristics
  - Items are associated with a value: priority
  - Provide access to one element at a time - the one with the highest priority
- Uses
  - Operating systems
  - Network management
    - Real time traffic usually gets highest priority when bandwidth is limited

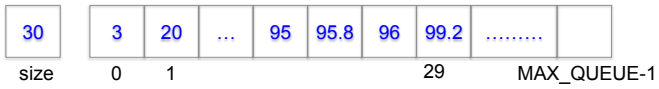
Priority Queue ADT Operations



1. Create an empty priority queue
   `createPQueue()`
2. Determine whether empty
   `pqIsEmpty():boolean`
3. Insert new item
   `pqInsert(in newItem:PQItemType) throws PQueueException`
4. Retrieve and delete the item with the highest priority
   `pqDelete():PQItemType`

PQ – Array Implementation



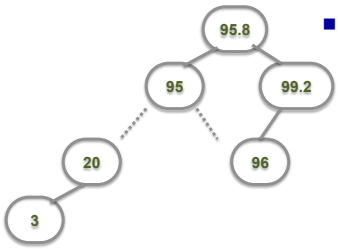


- ArrayList ordered by priority
  - Find the correct position for the insertion
  - Shift the array elements to make room for the new item

PQ – Reference-based Implementation

- Reference-based implementation
  - Sorted in descending order
    - Highest priority value is at the beginning of the linked list
    - `pqDelete` returns the item that `psHead` references and changes `pqHead` to reference the next item.
    - `pqInsert` must traverse the list to find the correct position for insertion.

PQ – BST Implementation

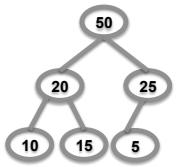


- Binary search tree
  - What is the highest value of the nodes?
  - Removing is easy
    - It has at most one child
  - You can use a balanced variation of the binary search tree.
- Other options?

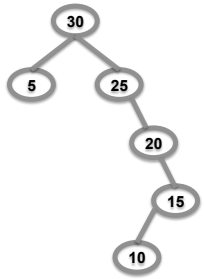
Heap - Definition

- A maximum heap (maxheap) is a ***complete*** **binary tree** that satisfies the following:
  - It is an empty tree
    or it has the heap property:
    - Its root contains a key greater or equal to the keys of its children
    - Its left and right subtrees are also heaps
- Implications of the heap property:
  - The root holds the maximum value (global property)
  - Values in descending order on every path from root to leaf
- **Heap is NOT a binary search tree.**

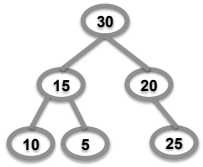
Examples



Satisfies heap property Complete



Satisfies heap property Not complete

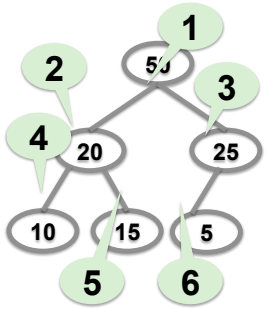


Does not satisfy heap property Not complete

Heap ADT

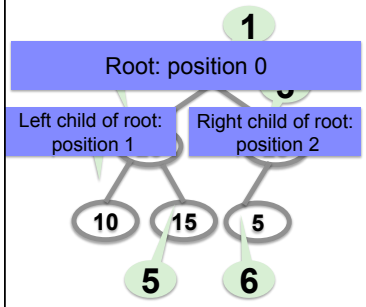
```
createHeap()  // create empty heap

heapIsEmpty():boolean
  // determines if empty

heapInsert(in newItem:HeapItemType)
  throws HeapException
  // inserts newItem based on its search key.
  // Throws exception if heap is full

heapDelete():HeapItemType
  // retrieves and then deletes heap's root
  // item which has largest search key
```

ArrayList Implementation



1	50
2	20
3	25
4	10
5	15
6	5

ArrayList Implementation



1	50
2	20
3	25
4	10
5	15
6	5

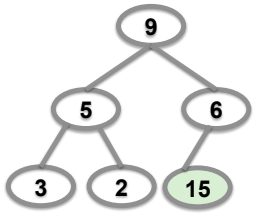
ArrayList Implementation

- Traversal items:
  - Root at position 0
  - Left child of position i at position 2i+1
  - Right child of position i at position 2(i+1)
  - Parent of position i at position $\lfloor (i-1)/2 \rfloor$

Heap Operations - `heapInsert`

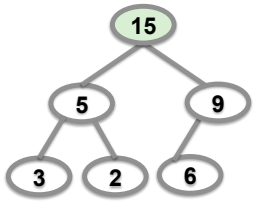
- **Step 1**: put a new value into first open position (maintaining completeness)
- **Step 2**: percolate values up
  - **Re-enforcing the heap property**
  - Swap with parent until in the right place

Insertion into a heap (Insert 15)



Insert 15
Trickle up
Trickle up

Insertion into a heap (Insert 15)



Heap Insert Pseudocode

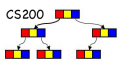
```
 // insert newItem into bottom of tree
items[size] = newItem
// percolate new item up to appropriate spot
place = size
parent = (place-1)/2
while (parent >= 0 and items[place] > items[parent])
  {
  swap items[place] and items[parent]
  place = parent
  parent = (place-1)/2
  }
increment size
```

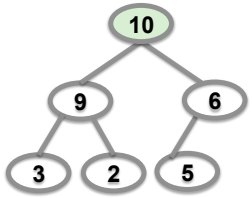
Part of the insert operation is often called `siftUp`

Heap operations – `heapDelete`

- **Step 1**: always remove value at root (Why?)
- **Step 2**: substitute with rightmost leaf of bottom level (Why?)
- **Step 3**: percolate/sift down
  - Swap with maximum child as necessary

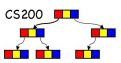
Deletion from a heap

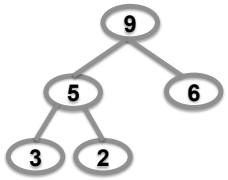




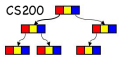
Delete 10
Place last node in root
Trickle down

Deletion from a heap



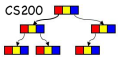


heapDelete Pseudocode




```
// return the item in root
rootItem = items[0]
//copy item from last node into root
items[0] = items[size-1]
size--
// restore the heap property
heapRebuild(items, 0, size)

return rootItem
```

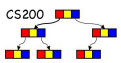

heapRebuild Pseudocode




```
heapRebuild(inout items:ArrayType, in root:integer,
            in size:integer)
  if (root is not a leaf) {
     child = 2 * root + 1   // left child
     if (root has right child) {
       rightChild = child + 1
       if (items[rightChild].getKey() >
             items[child].getKey()) {
           child = rightChild
       }
     } // larger child
     if (items[root].getKey() < items[child].getKey()) {
       swap items[root] and items[child]
       heapRebuild(items, child, size)
  }
}
```

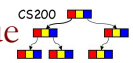
heapRebuild is also called siftDown

Array-based Heaps: Complexity



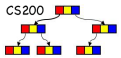
	Average	Worst Case
insert	O(log n)	O(log n)
delete	O(log n)	O(log n)

Heap versus BST for PriorityQueue



- BST can also be used to implement a priority queue
- How does worst case complexity compare?
- How does average case complexity compare?
- What if you know the maximum needed size for the PriorityQueue?

Small number of priorities

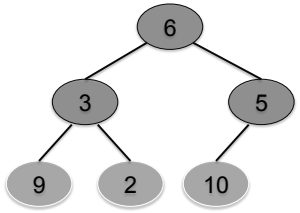


- A heap of queues with a queue for each priority value.

HeapSort

- Algorithm
  - Insert all elements (one at a time) to a heap
  - Iteratively delete them
    - Removes minimum/maximum value at each step
- Computational complexity?

Repeat n times of insert operation?



Question : Can we reduce the number of operations?

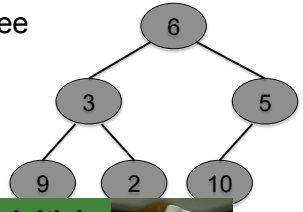
HeapSort

- Alternative method (in-place):
  - Create a heap out of the input array:
    - Consider the input array as a complete binary tree
    - Create a heap by iteratively expanding the portion of the tree that is a heap
      - Start from the leaves, which are semi-heaps
      - Move up to next level calling `heapRebuild` with each parent
  - Iteratively swap the root item with last item in unsorted portion and rebuild

Build initial tree

6	3	5	9	2	10

- Begin with the root
- Left to right down this tree

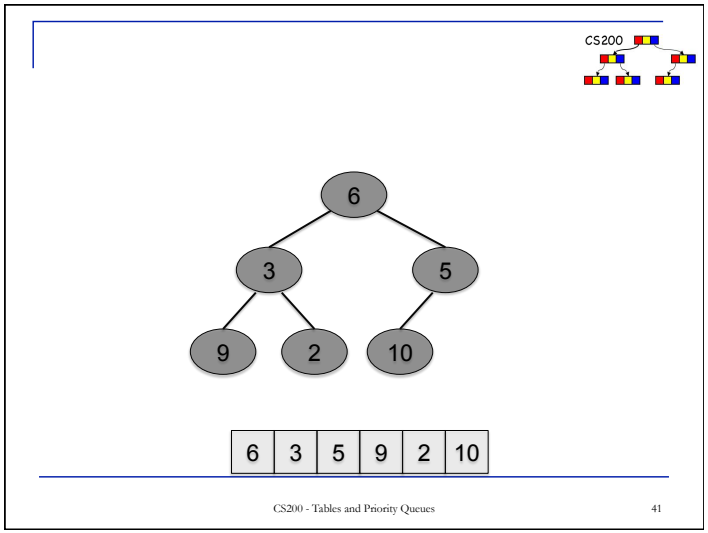


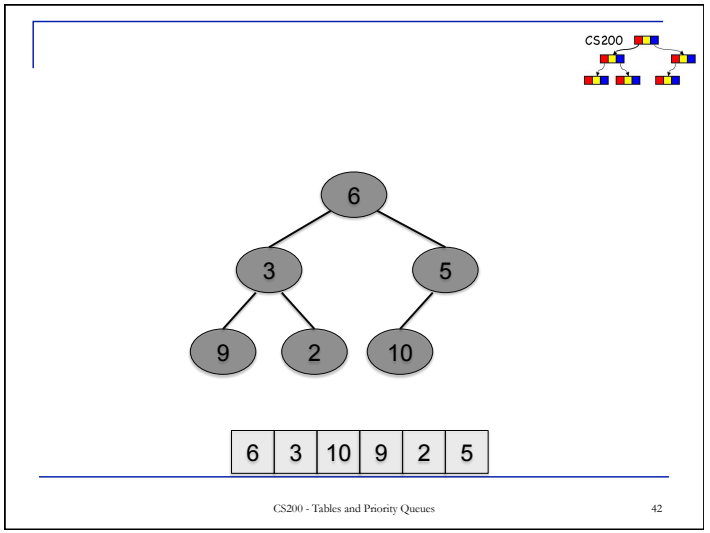
Question : To build this initial heap, how many operations are needed?
A. 3 B. 4 C. 5 D. 6

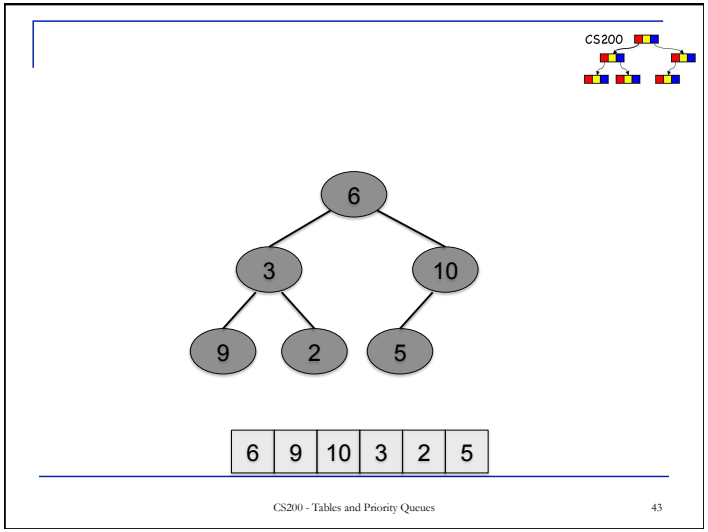
Transform tree into a heap

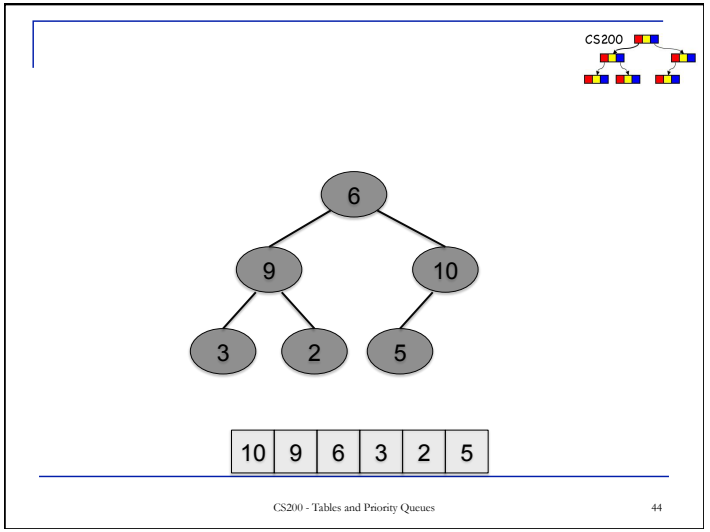
```
for (index = n -1 down to 0)
    //Assertion: the tree rooted at index is a semiheap
    heapRebuild(anArray, index x)
    //Assertion: the tree rooted at index is a heap
```

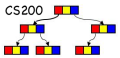
- Call `heapRebuild()` on the leaves from right to left
- Move up the tree

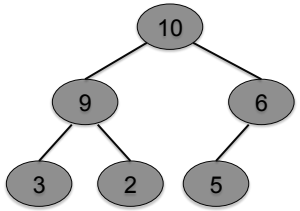

CS200
6
3
5
9
2
10
6 3 5 9 2 10
CS200 - Tables and Priority Queues
41


CS200
6
3
5
9
2
10
6 3 10 9 2 5
CS200 - Tables and Priority Queues
42


CS200
6
3
10
9
2
5
6 9 10 3 2 5
CS200 - Tables and Priority Queues
43

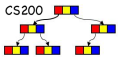

CS200
6
9
10
3
2
5
10 9 6 3 2 5
CS200 - Tables and Priority Queues
44

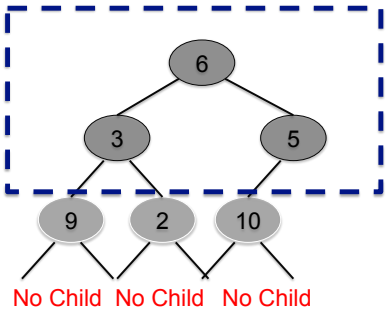




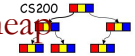
10	9	6	3	2	5

Do we need n steps?

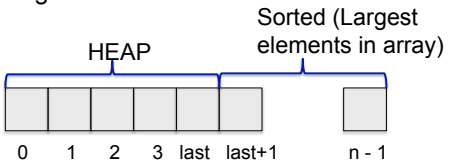




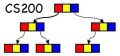
After transforming the array into a heap



- Heapsort partitions the array into two regions
  - Heap region
  - sorted region



Sorted : n-1 (5~)



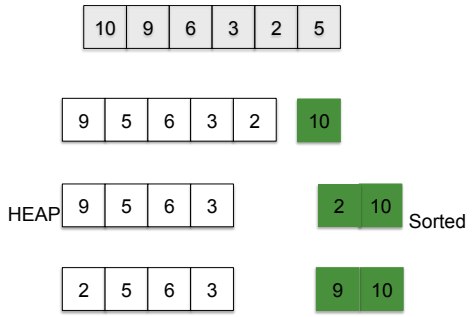
10	9	6	3	2	5

HEAP | 10 | 9 | 6 | 3 | 2 | 5 | Sorted

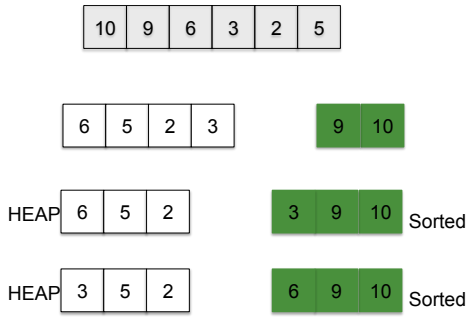
HEAP | 10 | 9 | 6 | 3 | 2 | 5 | Sorted

HEAP | 5 | 9 | 6 | 3 | 2 | 10 | Sorted

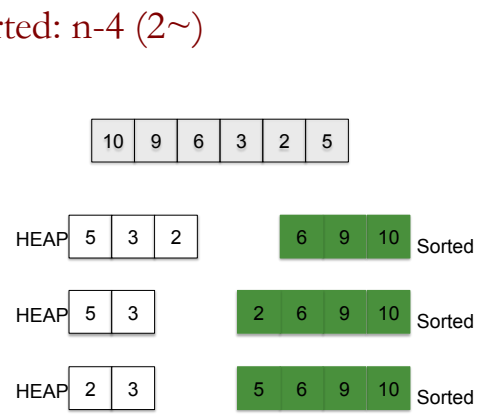
Sorted: n-2 (4~)



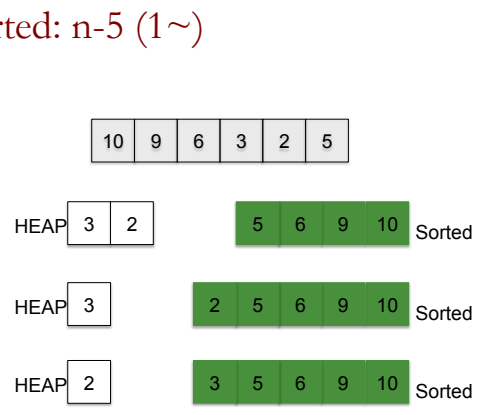
Sorted: n-3 (3~)



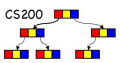
Sorted: n-4 (2~)

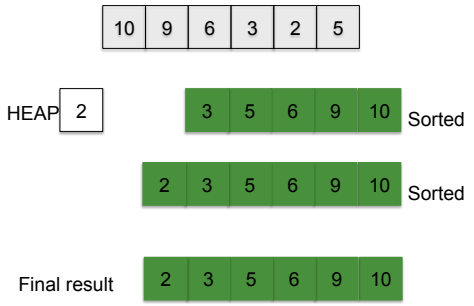


Sorted: n-5 (1~)

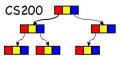


Sorted :n-6 (0~)



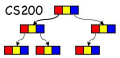


HeapSort Pseudocode




```
heapSort(ourItems:ArrayList, n:integer)
  // First step: build heap out of the input array
  for (index = n - 1 down to 0) {
      // Invariant: the tree rooted at index is a
      // semiheap
      // semiheap: tree where the subtrees of the
      //   root are heaps
      heapRebuild(ourItems, index, n)
      // The tree rooted at index is a heap
  }
```


HeapSort Pseudocode




```
heapSort(ourItems:ArrayList, n:integer)
  for (index = n-1 down to 0) {
      heapRebuild(ourItems, index, n)
  }
  last = n -1 // initialize the regions
  for (step = 1 through n) {
      swap ourItems[0] and ourItems[last]
      decrement last
      heapRebuild(ourItems, 0, last)  }
```