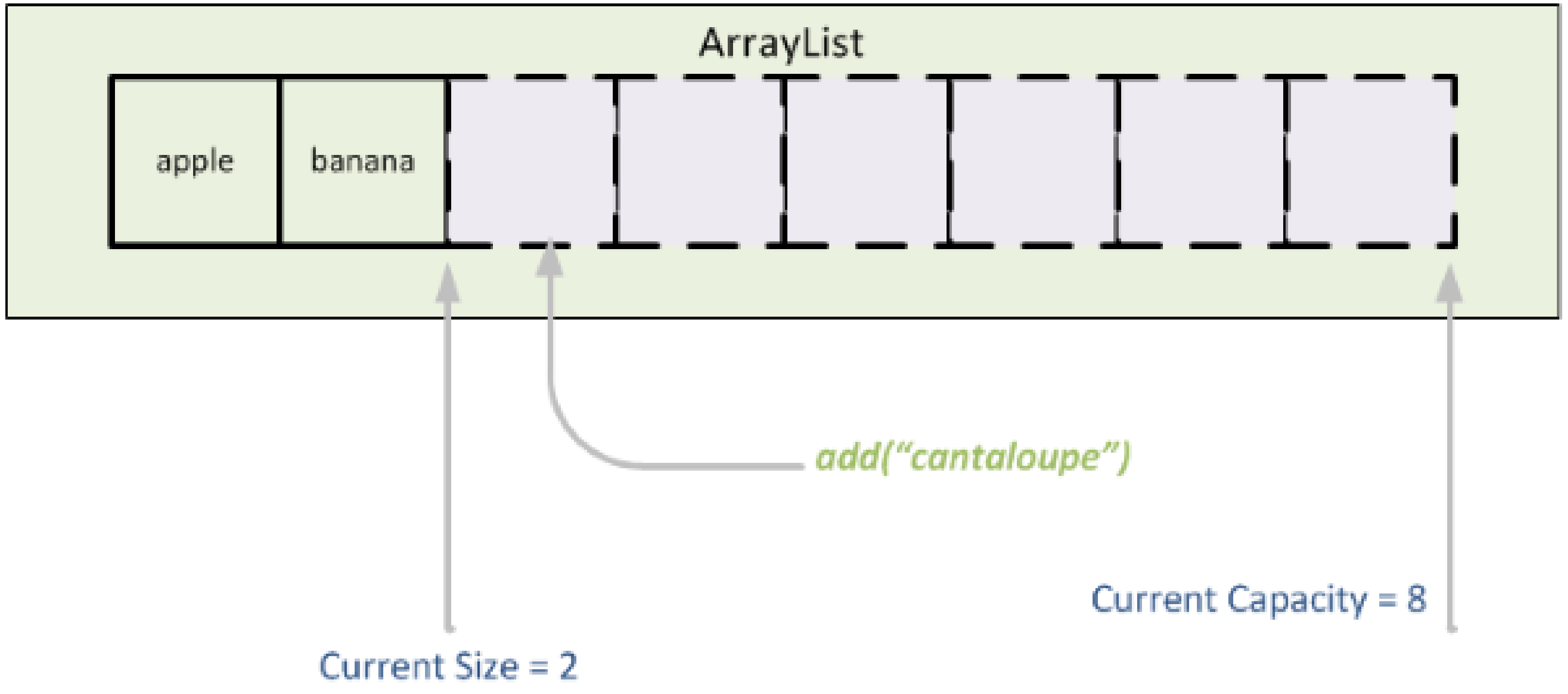
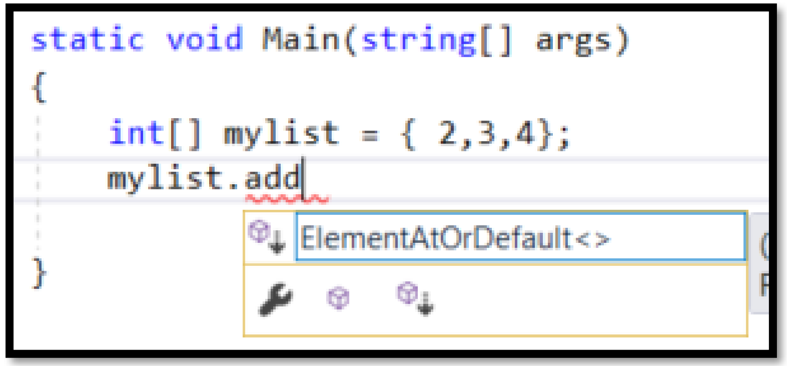
M6: Hash Tables, Array Lists + Extra (Heaps)

Summer 2019 – CSC 395 – ST: Algorithm & Data Structure Concepts

ARRAY-BASED LISTS (ARRAY LISTS)



ARRAY-BASED LISTS



```
static void Main(string[] args)
{
    int[] mylist = { 2,3,4};
    mylist.add
}
```

- Can you add a new value to an existing array?
- To implement a list using arrays, we allocate more space than is necessary.
- The array is used until it is full.
- Once an array is full, either all new insertions fail until an item is removed or the array must be reallocated.
  - The reallocation typically involves creating a larger array, copying over the old data, and making this the array.
  - As the process of growing an array requires the typically copying of the entire array, this is not something you want to do often
  - That is, you do not grow the array a few elements at a time but in large chunks instead.
  - The exact number of elements to grow by is implementation specific.

ARRAY-BASED LISTS – START IN CLASS

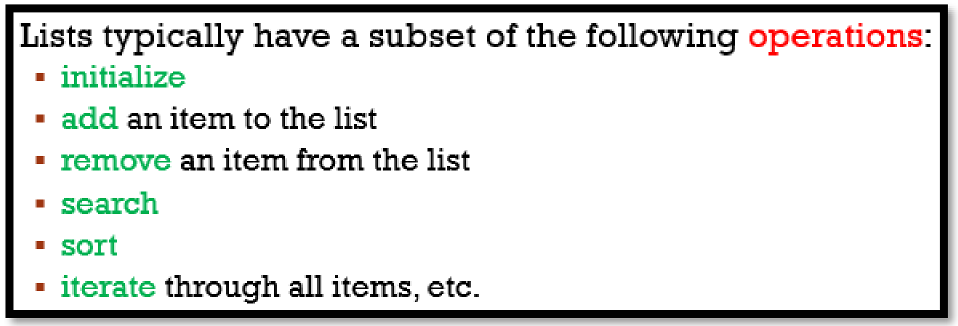
- Create a class MyArrayList that is an array-based list and implements:
  - isEmpty()
  - isFull()
  - size()
  - printAll()
  - addBack(value) ←add(value)
  - addFront(value)
  - insert(index)
  - deleteBack()
  - deleteFront()
  - delete(index)
  - contains(value)
  - clear()
  - sort()

what is the big Oh for each of these operations?

←think of these as user interface

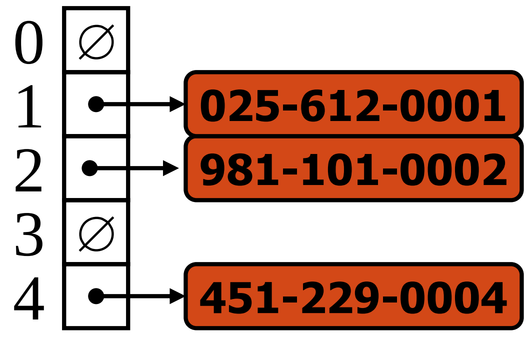
methods define the behavior

they usually operate on data


Lists typically have a subset of the following operations:
- initialize
- add an item to the list
- remove an item from the list
- search
- sort
- iterate through all items, etc.

HASH TABLES

Hash Tables



HASH FUNCTIONS AND HASH TABLES

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
  - Example:
    $h(x) = x \bmod N$
    is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h-------------------> hash function
  - Array (called table) of size N-----> bucket array
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

HASH FUNCTIONS AND HASH TABLES

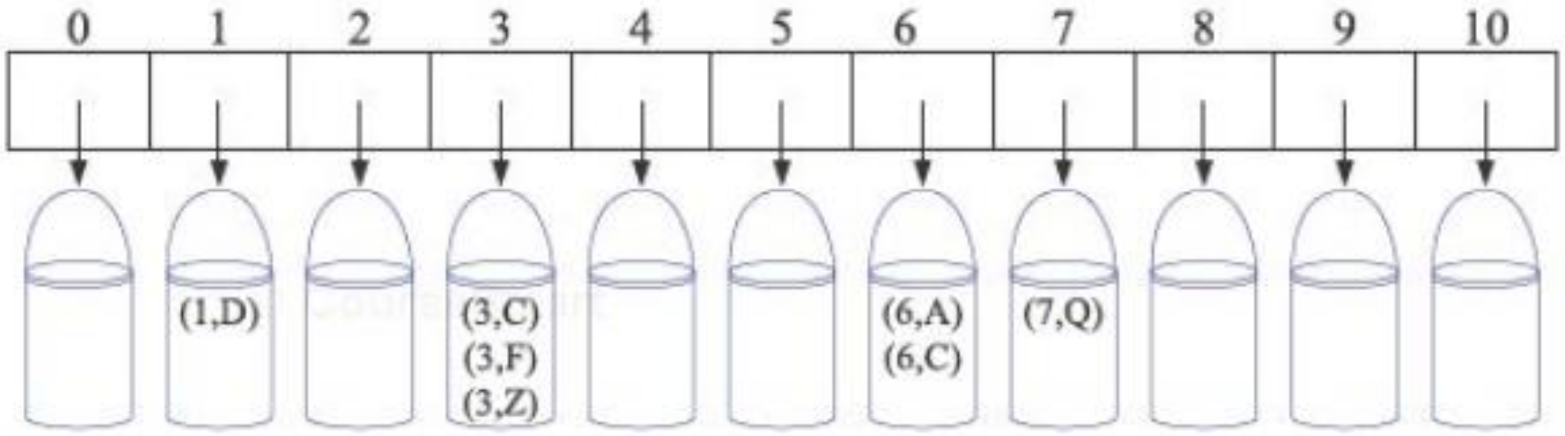


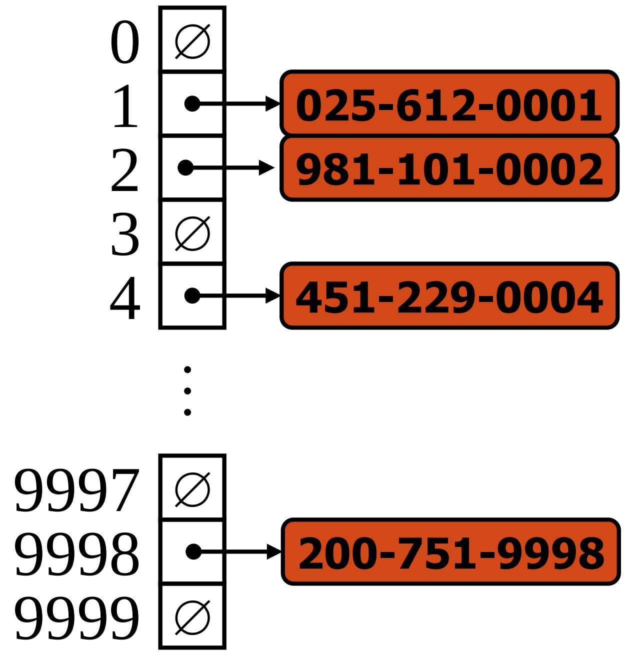
Figure 9.2: A bucket array of size 11 for the entries (1,D), (3,C), (3,F), (3,Z), (6,A), (6,C), and (7,Q).

HASH FUNCTIONS AND HASH TABLES

- If our keys are unique integers in the range [0,N-1], then each bucket holds at most one entry. Thus, searches, insertions, and removals in the bucket array take O(1) time.
- If there are two or more keys with the same hash value, then two different entries will be mapped to the same bucket in A. In this case, we say that a **collision** has occurred.
- A hash function is "good" if it maps the keys in our map in such a way as to minimize collisions as much as possible.

EXAMPLE

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $\boldsymbol{h}(\boldsymbol{x}) = \text{last four digits of } \boldsymbol{x}$



HASH FUNCTIONS

- A hash function is usually specified as the composition of two functions:

  Hash code:
  h_1: keys $\rightarrow$ integers

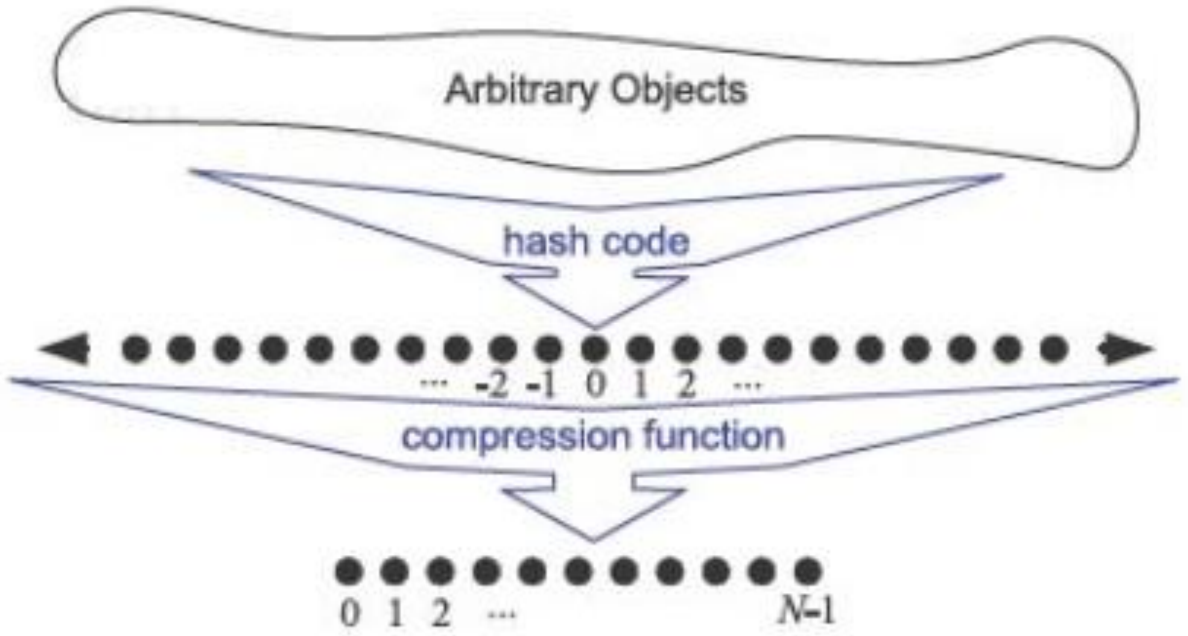
  Compression function:
  h_2: integers $\rightarrow$ $[0, N-1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$h(x) = h_2(h_1(x))$$

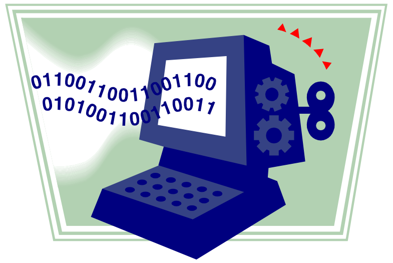
- The goal of the hash function is to "disperse" the keys in an apparently random way

HASH FUNCTIONS



The two parts of a hash function: hash code and compression function.

HASH CODES



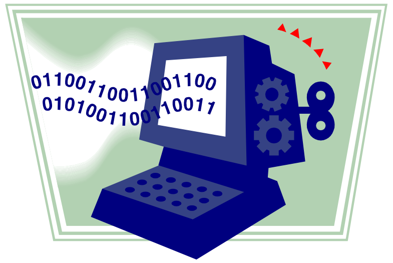
- we want the set of hash codes assigned to our keys to avoid collisions as much as possible.

- the hash code we use for a key k should be the same as the hash code for any key that is equal to k.

- The hash codes described below are based on the assumption that the number of bits of each type is known. This information is provided in the standard include file <limits>. This include file defines a templated class numeric_limits. Given a base type T (such as **char**, **int**, or **float**), the number of bits in a variable of type T is given by

  " numeric_limits<T>. digits "

HASH CODES



- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in C++)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in C++)
  - Not good for character strings (spot, tops,pots … would collide) or other variable-length objects

Hash Codes (cont.)

- Polynomial accumulation:
  - Takes into consideration the position of the components
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $$a_0\, a_1\, \ldots\, a_{n-1}$$
  - We evaluate the polynomial
    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$$
    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Cyclic Shift Hash Codes …

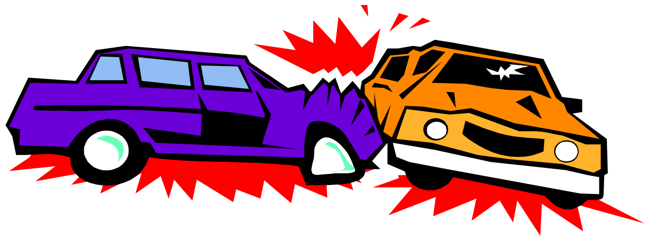
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z)$$
    $$(i = 1, 2, \ldots, n-1)$$
- We have $p(z) = p_{n-1}(z)$

COMPRESSION FUNCTIONS

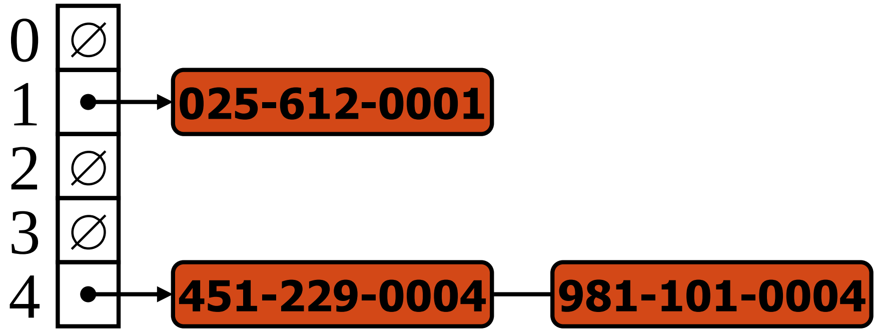
- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory …
  - Hashing {200, 205, 210, 215, 220, … , 600} to a bucket array of size 100 then each hash code collides with three others. But if this same set of keys is similarly hashed to a bucket array of size 101, then there are no collisions.

- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b
  - N is a prime number

COLLISION HANDLING



- Collisions occur when different elements are mapped to the same cell



- Separate Chaining: let each cell in the table point to a linked list of entries that map there
  - Separate chaining is simple, but requires additional memory outside the table

- Open addressing: the colliding item is placed in a different cell of the table

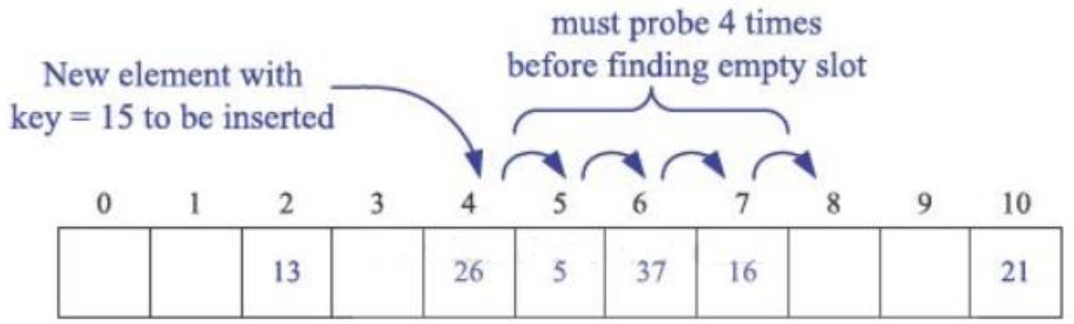

New element with key = 15 to be inserted
must probe 4 times before finding empty slot
0
1
2
3
4
5
6
7
8
9
10
13
26
5
37
16
21

Figure 9.5: An insertion into a hash table using linear probing to resolve collisions. Here we use the compression function $h(k) = k \bmod 11$.

MAP WITH SEPARATE CHAINING

Delegate operations to a list-based map at each cell:

```
**Algorithm** find(k):
return A[h(k)].find(k)

Algorithm put(k,v):
p = A[h(k)].put(k,v)
n = n + 1
return p

Algorithm erase(k):
A[h(k)].erase(k)
n=n - 1
```

MAP WITH SEPARATE CHAINING

- Assuming we use a good hash function to index the n entries of our map in a bucket array of capacity N, we expect each bucket to be of size $\lceil n/N \rceil$. This value, called the **load factor** of the hash table, should be bounded by a small constant, preferably below 1.

- The expected running time of operations find, put, and erase in a map implemented with a hash table that uses this function is O(n/N). Thus, we can implement these operations to run in O(1) expected time provided n is O(N).

MAP WITH SEPARATE CHAINING

Delegate operations to a list-based map at each cell:

```
**Algorithm** find(k):
return A[h(k)].find(k)
// delegates the find(k) to the list-based map at A[h(k)]

Algorithm put(k,v):
p = A[h(k)].put(k,v)
n = n + 1
return p
// delegates the put to the list-based map at A[h(k)]

Algorithm erase(k):
A[h(k)].erase(k)
n=n - 1
// delegates the erase(k) to the list-based map at A[h(k)]
```

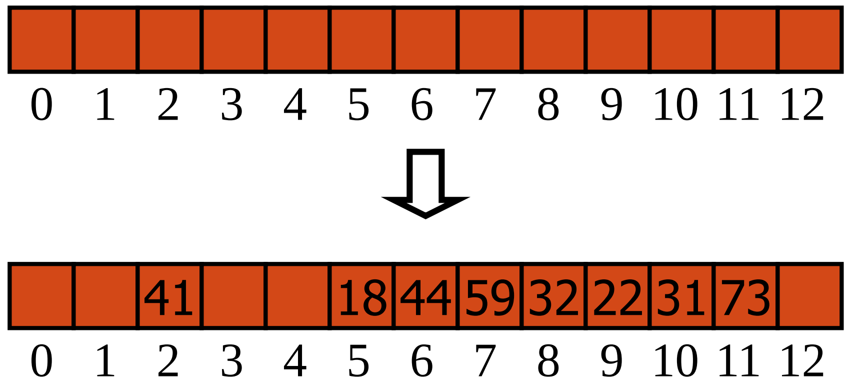
OPEN ADDRESSING

- This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to deal with collisions.
- Open addressing requires that the load factor is always at most 1 and that entries are stored directly in the cells of the bucket array itself.
- Open addressing: **the colliding item is placed in a different cell of the table**

LINEAR PROBING

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



LINEAR PROBING

- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- to perform a search, followed by either a replacement or insertion, we must examine consecutive buckets, starting from A[h(k)], until we either find an entry with key equal to k or we find an empty bucket.

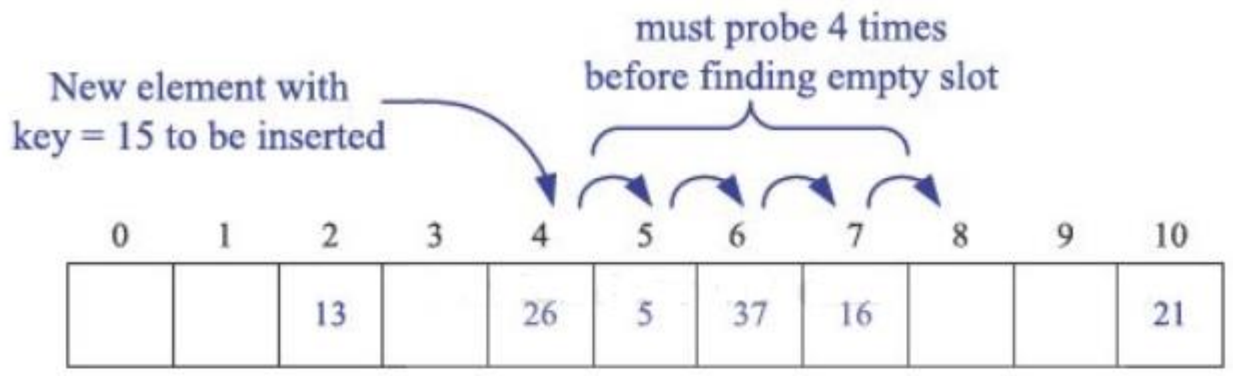
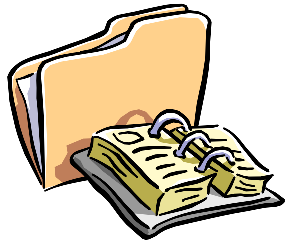

New element with key = 15 to be inserted
must probe 4 times before finding empty slot
0
1
2
3
4
5
6
7
8
9
10
13
26
5
37
16
21

Figure 9.5: An insertion into a hash table using linear probing to resolve collisions. Here we use the compression function $h(k) = k \bmod 11$.

SEARCH WITH LINEAR PROBING



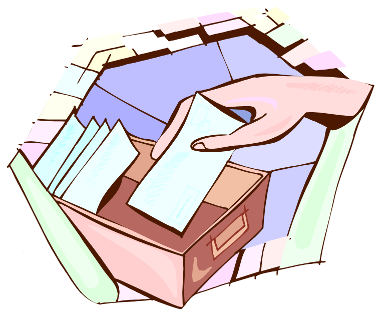
- Consider a hash table A that uses linear probing
- find(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm find(k)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return null
        else if $c.key() = k$
            return $c.value()$
        else
            $i \leftarrow (i + 1) \bmod N$
            $p \leftarrow p + 1$
    until $p = N$
    return null
```

UPDATES WITH LINEAR PROBING

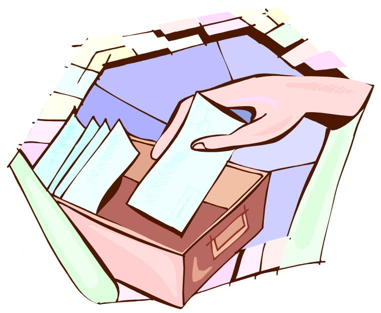
- To handle insertions and deletions, we introduce a special object, called *AVAILABLE*, which replaces deleted elements
- erase(***k***)
  - We search for an entry with key ***k***
  - If such an entry (***k, o***) is found, we replace it with the special item *AVAILABLE* and we return element ***o***
  - Else, we return ***null***
- put(***k, o***)
  - We throw an exception if the table is full
  - We start at cell ***h***(***k***)
  - We probe consecutive cells until one of the following occurs
    - A cell ***i*** is found that is either empty or stores *AVAILABLE*, or
    - *N* cells have been unsuccessfully probed
  - We store (***k, o***) in cell ***i***

QUADRATIC PROBING



- involves iteratively trying the buckets A[(i+f(j))mod N], for j = 0,1,2, . . ., where f (j) = j^2, until finding an empty bucket.
- the quadratic-probing strategy complicates the removal operation, but it does avoid the kinds of clustering patterns that occur with linear probing.
- Nevertheless, it creates its own kind of clustering, called **secondary clustering**, where the set of filled array cells "bounces" around the array in a fixed pattern.
- Even if N is prime, this strategy may not find an empty slot if the bucket array is at least half full.

DOUBLE HASHING



- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

  $$(i + jd(k)) \bmod N$$

  for $j = 0,\ 1, \ldots, N - 1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression function for the secondary hash function:

  $$d_2(k) = q - k \bmod q$$

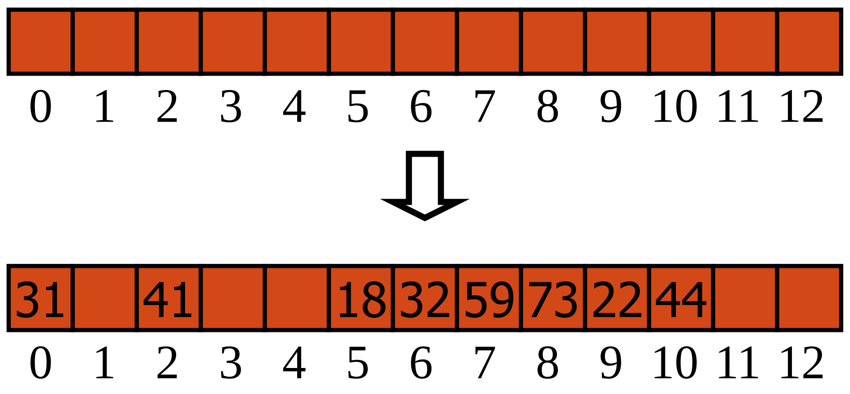
  where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are

  $$1, 2, \ldots, q$$

EXAMPLE OF DOUBLE HASHING

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

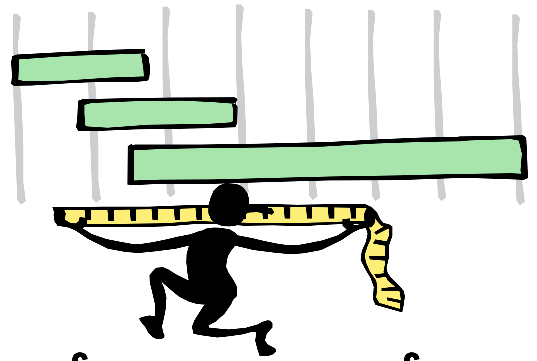
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



…

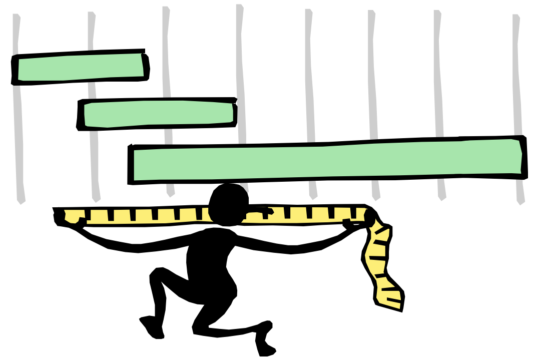
- These open-addressing schemes save some space over the separate- chaining method
- They are not necessarily faster. In experimental and theoretical analyses, the chaining method is either competitive or faster than the other methods, depending on the load factor of the bucket array.
- If memory space is not a major issue, the collision-handling method of choice seems to be separate chaining.

Performance of Hashing



- The load factor $\alpha = n/N$ affects the performance of a hash table
- Experiments and average-case analyses suggest that we should maintain $\alpha < 0.5$ for the open-addressing schemes and we should maintain $\alpha < 0.9$ for separate chaining.
- If our hash function is good, then we expect the hash function values to be uniformly distributed in the range [0, N - 1]. Thus, to store n items in our map, the expected number of keys in a bucket would be $\lceil n/N \rceil$ at most, which is O(1), if n is O(N).
- Applications of hash tables:
  - small databases, compilers, browser caches

Performance of Hashing



- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is
  $1 / (1 - \alpha)$
- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

THE FOLLOWING SLIDES ARE JUST FOR YOU, NOT NEEDED FOR THE EXAM

SOME RESOURCES

- https://legacy.gitbook.com/book/cathyatseneca/data-structures-and-algorithms/details
- See chapter 4 from http://people.cs.vt.edu/~shaffer/Book/C++3e20130328.pdf

SOME RESOURCES (2)

- suggested videos:
  - merge sort: https://www.youtube.com/watch?v=KF2j-9iSf4Q
  - merge sort in C#: https://www.youtube.com/watch?v=5NEQw1fhYsY&index=5&list=PLnREHExby2fHJnbry8Xx6jmYW8QUTSgOI
  - merge sort: https://www.youtube.com/watch?v=XaqR3G_NVoo
  - quicksort: https://www.youtube.com/watch?v=y_G9BkAm6B8
  - quicksort: https://www.youtube.com/watch?v=SLauY6PpjW4
  - heaps: https://www.youtube.com/watch?v=t0Cq6tVNRBA
  - heaps: https://www.youtube.com/watch?v=v1YUApMYXO4
  - heaps: https://www.youtube.com/watch?v=6NB0GHY11Iw
  - heap sort: https://www.youtube.com/watch?v=D_B3HN4gcUA
  - heap sort: https://www.youtube.com/watch?v=onlhnHpGgC4
  - heap sort: https://www.youtube.com/watch?v=mAO8LpQ6uGQ
- visualization:
  - heap: http://www.cs.armstrong.edu/liang/animation/web/Heap.html

RECURSION – REVIEW IF TIME

- A function that calls itself is a recursive function.
- Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.
- The simpler-to-solve problem is known as the **base case**
- Recursive calls stop when the base case is reached
- Recursion uses a process of breaking a problem down into smaller problems until the problem can be solved
- If time, please ask me to show the Hanoi Towers problem & solution.

RECURSION VS. ITERATION

- Benefits (+), disadvantages(-) for recursion:
  - + Models certain algorithms most accurately
  - + Results in shorter, simpler functions
  - – May not execute very efficiently (a lot of overhead…)

- Benefits (+), disadvantages(-) for iteration:
  - + Executes more efficiently than recursion
  - – Often is harder to code or understand

HEAP DEFINITION

- The **HeapSort** algorithm makes use of a data structure called a **heap**

- A **heap** is a binary tree with the following properties:
  - It is complete meaning that each row (except the last one) must be filled in
  - Each node is greater than or equal to any of its children. ←called the heap property
    - This way the root (top node) always contains the largest value. ← hence the name max heap
    - Alternatively one could build the heap so that the smallest value is always on top ← hence the name min heap
    - priority queues are often referred to as "heaps"

- Heaps are usually built using arrays rather than using node references such as the linked list.
  - So we think of heaps as binary trees, but we build them using arrays (or array lists)

HEAP APPLICATIONS

- Heap Data Structure is generally taught with Heapsort.
  - Heapsort algorithm has limited uses because Quicksort is better in practice.
  - Nevertheless, the Heap data structure itself is enormously used. Following are some uses other than Heapsort.

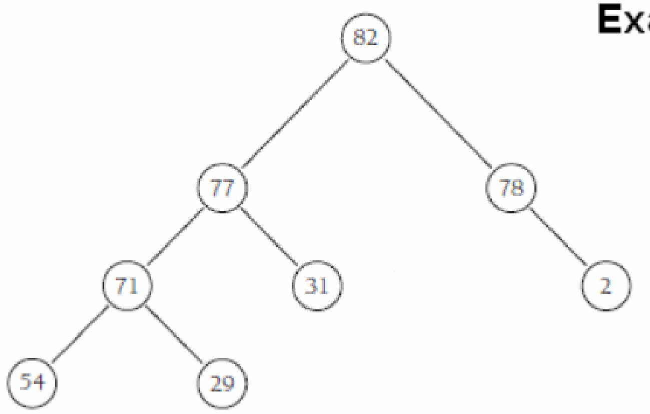
- Priority Queues:
  - Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time.
  - Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm.

- Order statistics:
  - The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.
    See https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/ **Time complexity:** O(n + klogn)

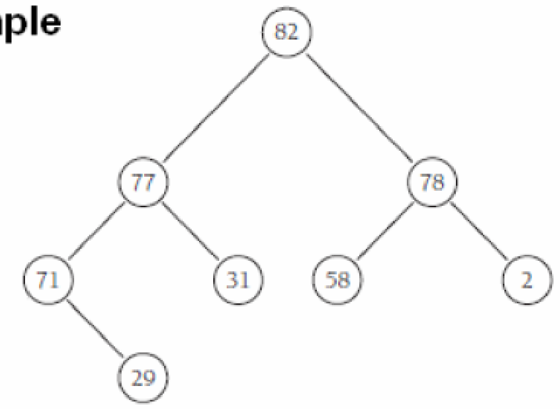
- Source: https://www.geeksforgeeks.org/applications-of-heap-data-structure/

COMPLETE BINARY TREE

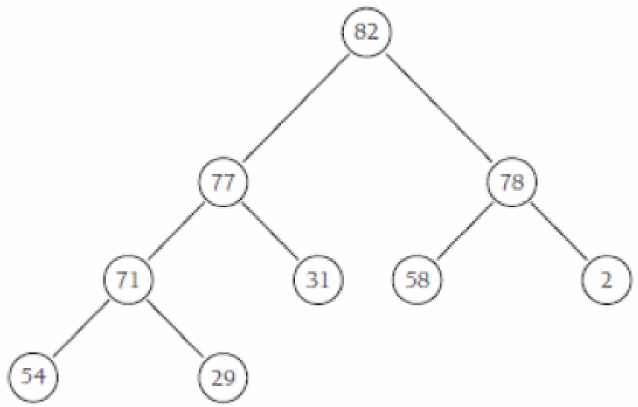
Example



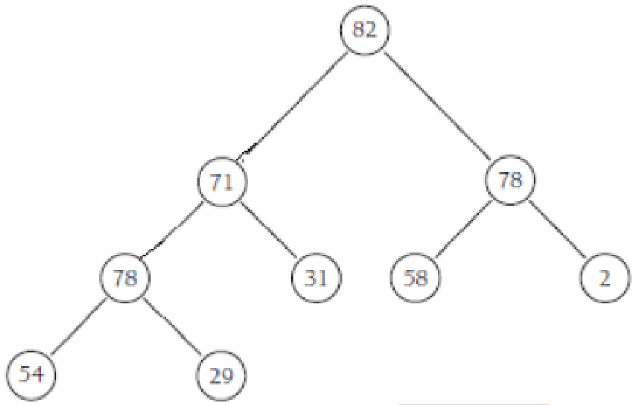
Not Complete



Not Complete



A Heap.

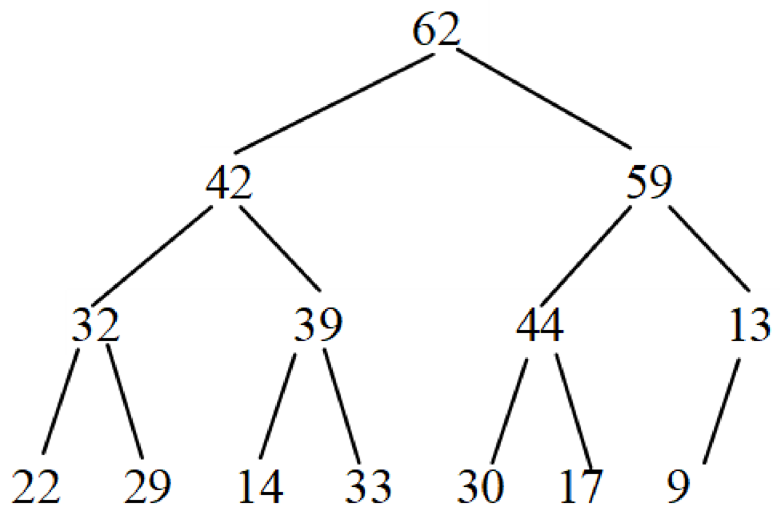


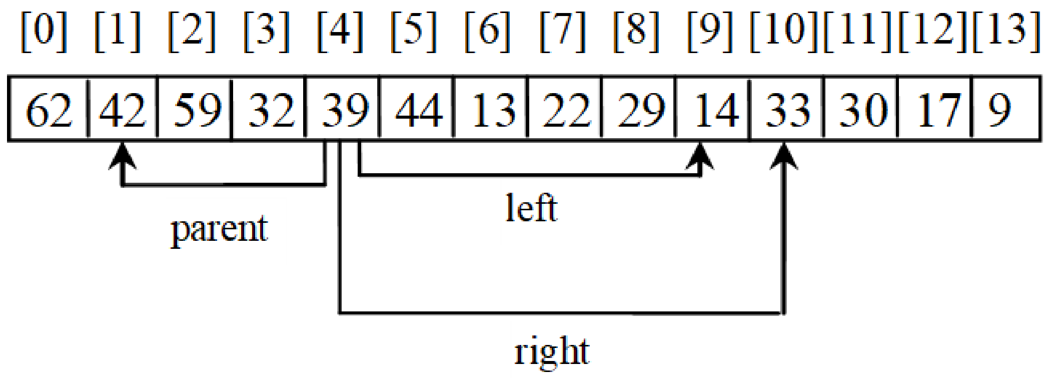
Not Heap

- A binary tree is *complete* if every level of the tree is full except that the last level may not be full and all the leaves on the last level are placed left-most.

HEAP REPRESENTATION

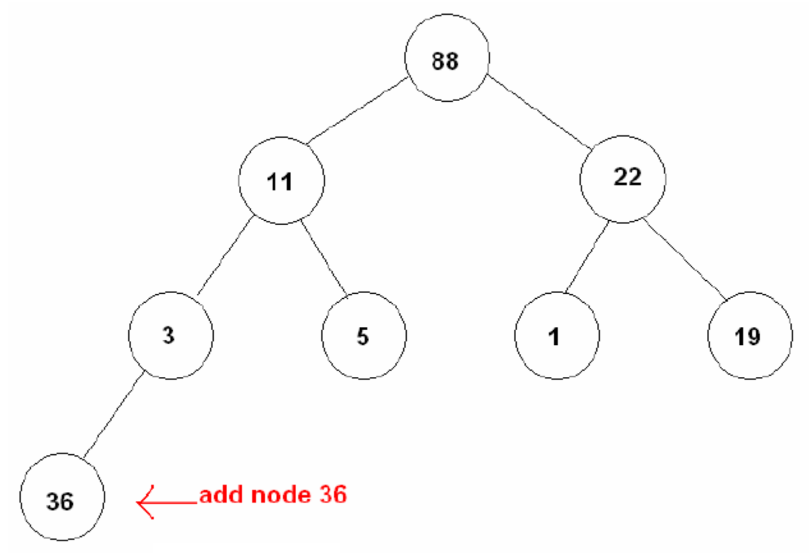
- For a node at position *i*, its left child is at position *2i+1* and its right child is at position *2i+2*, and its parent is *(i-1)/2*.
- For example, the node for element 39 is at position 4, so its left child (element 14) is at 9 (*2*4+1*), its right child (element 33) is at 10 (*2*4+2*), and its parent (element 42) is at 1 (*(4-1)/2*).

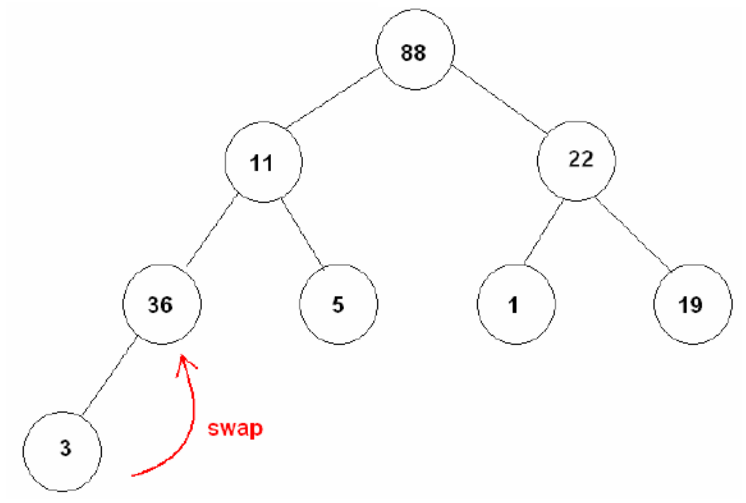


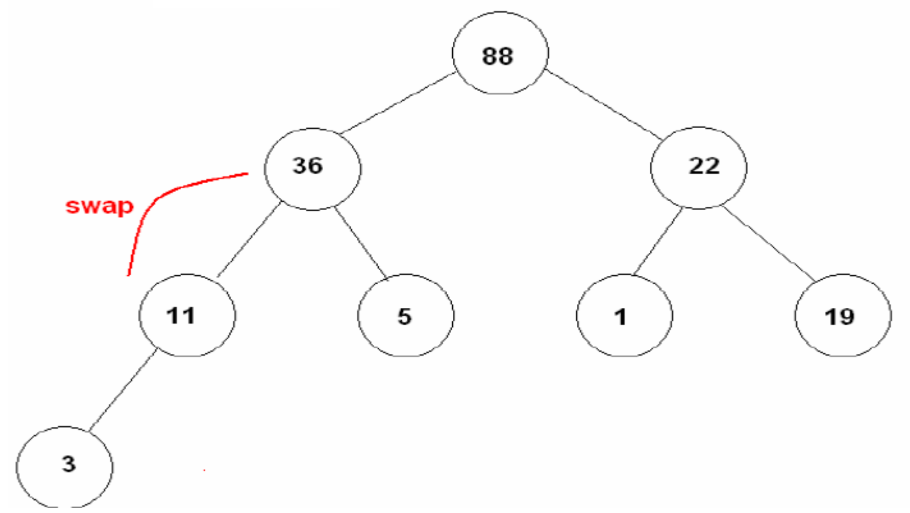


ADDING ELEMENTS TO THE HEAP

- Adding 36 to a non-empty heap

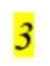




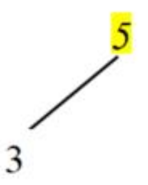


ADDING ELEMENTS TO THE HEAP (2)

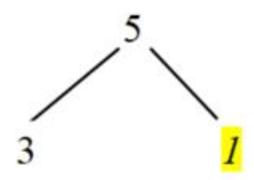
- Adding 3, 5, 1, 19, 11, and 22 to a heap, initially empty



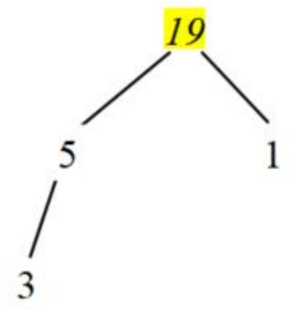
(a) After adding 3



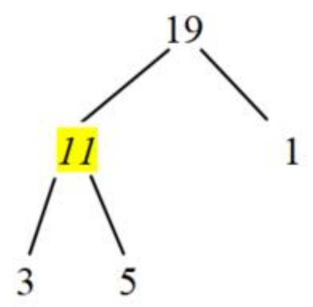
(b) After adding 5



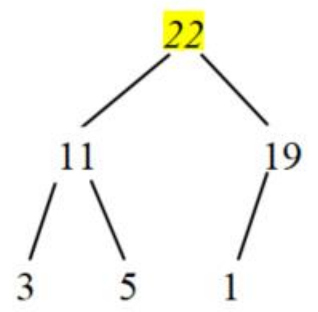
(c) After adding 1



(d) After adding 19



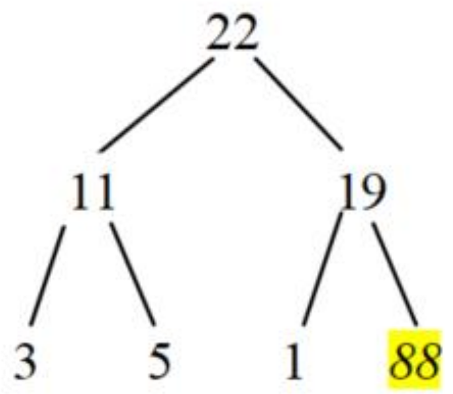
(e) After adding 11



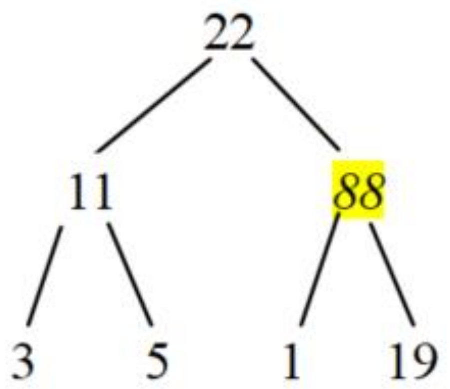
(f) After adding 22

REBUILD THE HEAP AFTER ADDING A NEW NODE

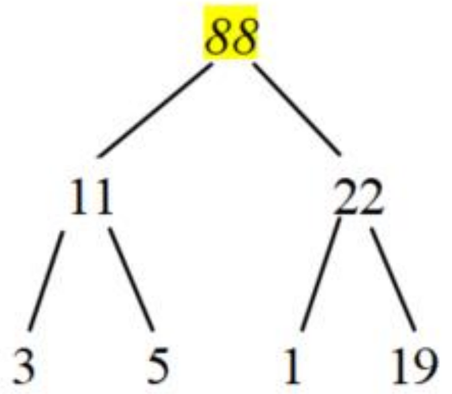
- Adding 88 to the heap



(a) Add 88 to a heap



(b) After swapping 88 with 19



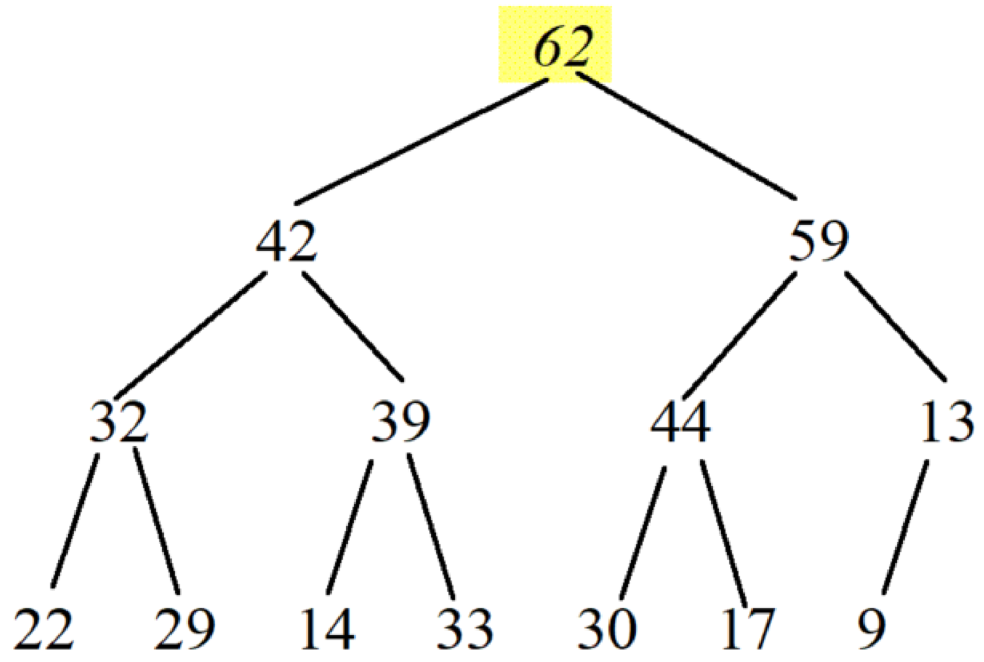
(b) After swapping 88 with 22

ADDING AN ELEMENT TO THE HEAP (3)

- A new node is always placed at the end of the array in an empty array element.
- The problem is that doing this will probably break the heap condition because the new node's data value may be greater than some of the nodes above it.
- To restore the array to the proper heap condition, we must shift the new node up until it reaches its proper place in the array.

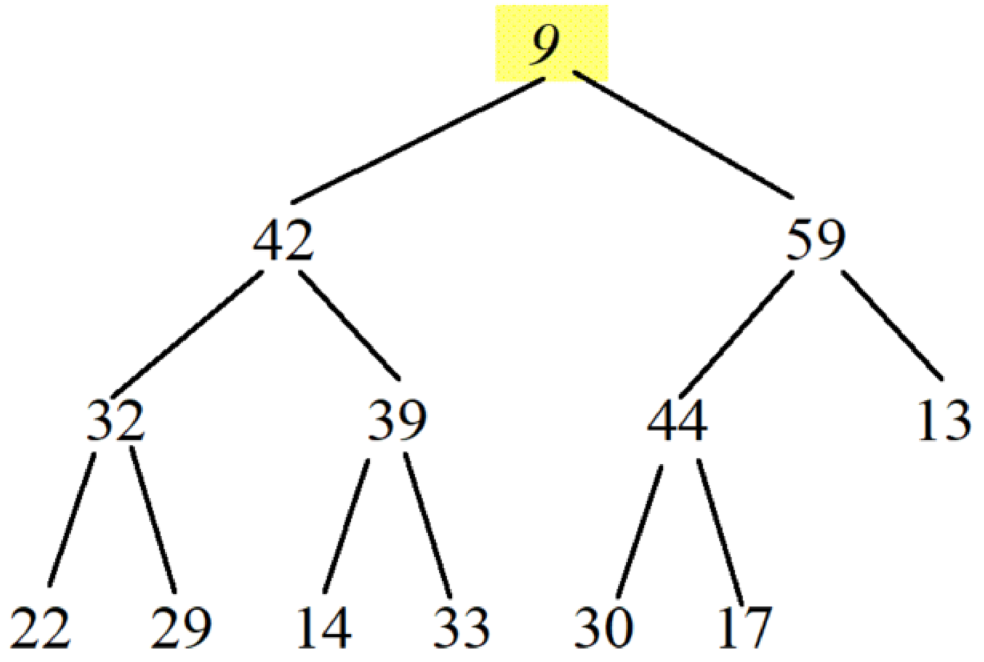
REMOVING THE ROOT AND REBUILD THE TREE

- Removing root 62 from the heap



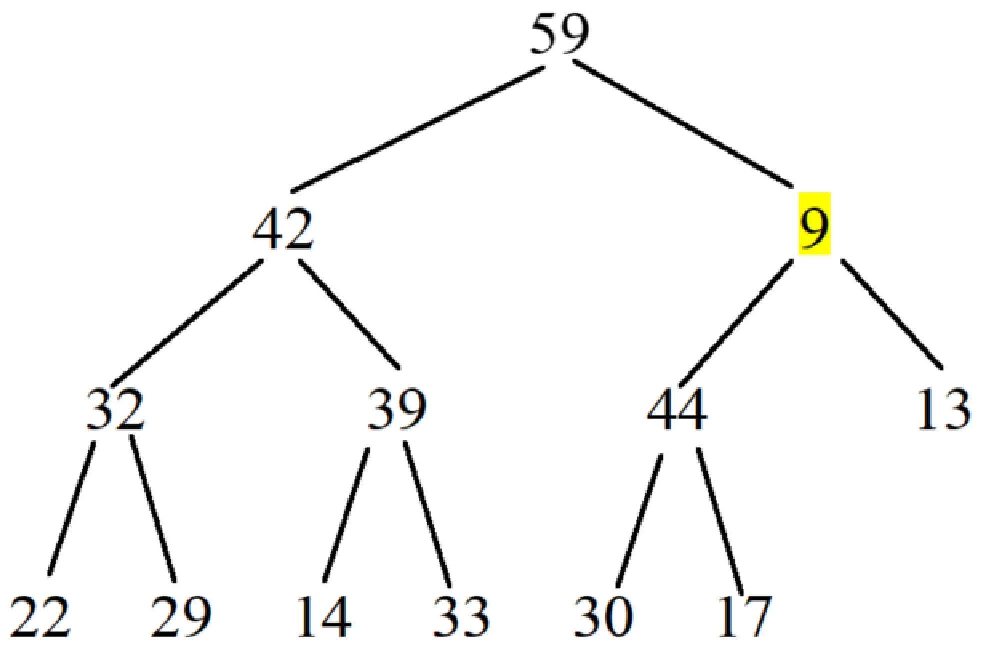
REMOVING THE ROOT AND REBUILD THE TREE

- Move 9 to root



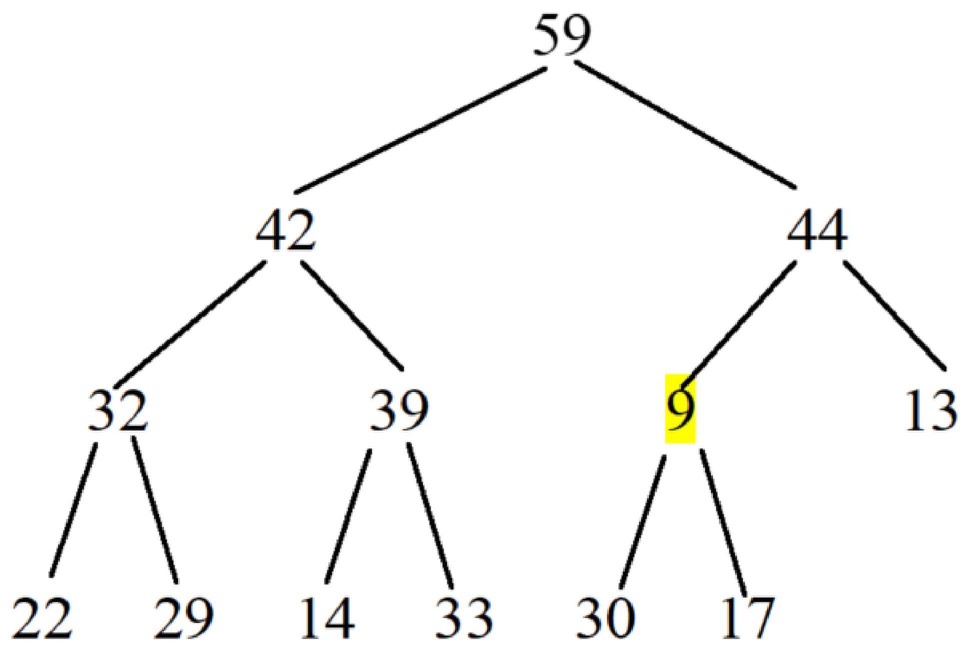
REMOVING THE ROOT AND REBUILD THE TREE

- Swap 9 with 59 (since 59 is the largest of the two child nodes)



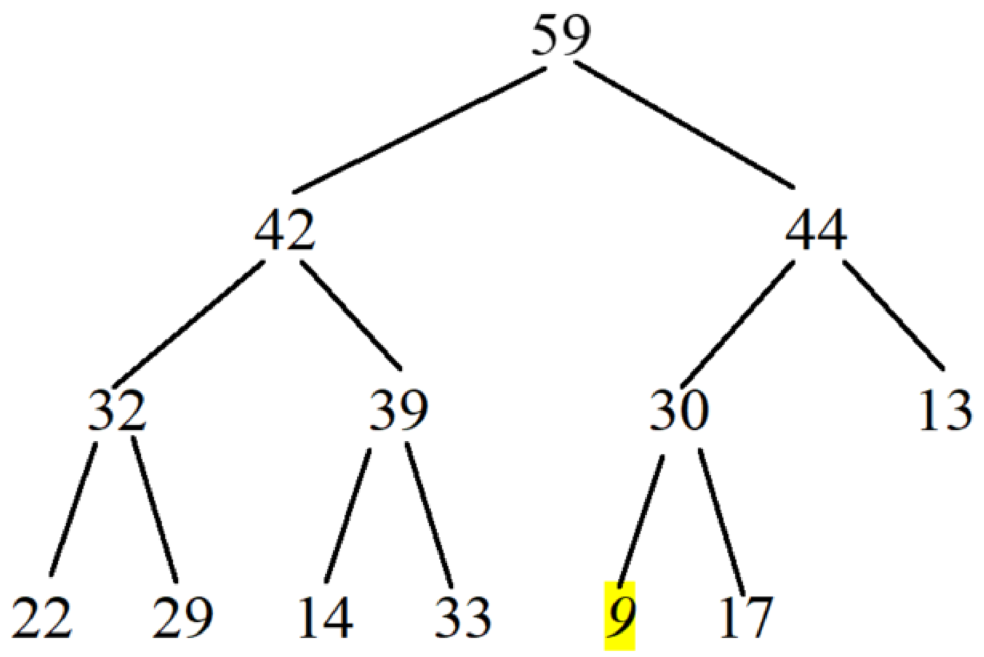
REMOVING THE ROOT AND REBUILD THE TREE

- Swap 9 with 44 (since 44 is the largest of the two child nodes)



REMOVING THE ROOT AND REBUILD THE TREE

- Swap 9 with 30 (since 30 is the largest of the two child nodes)



HEAP ADT

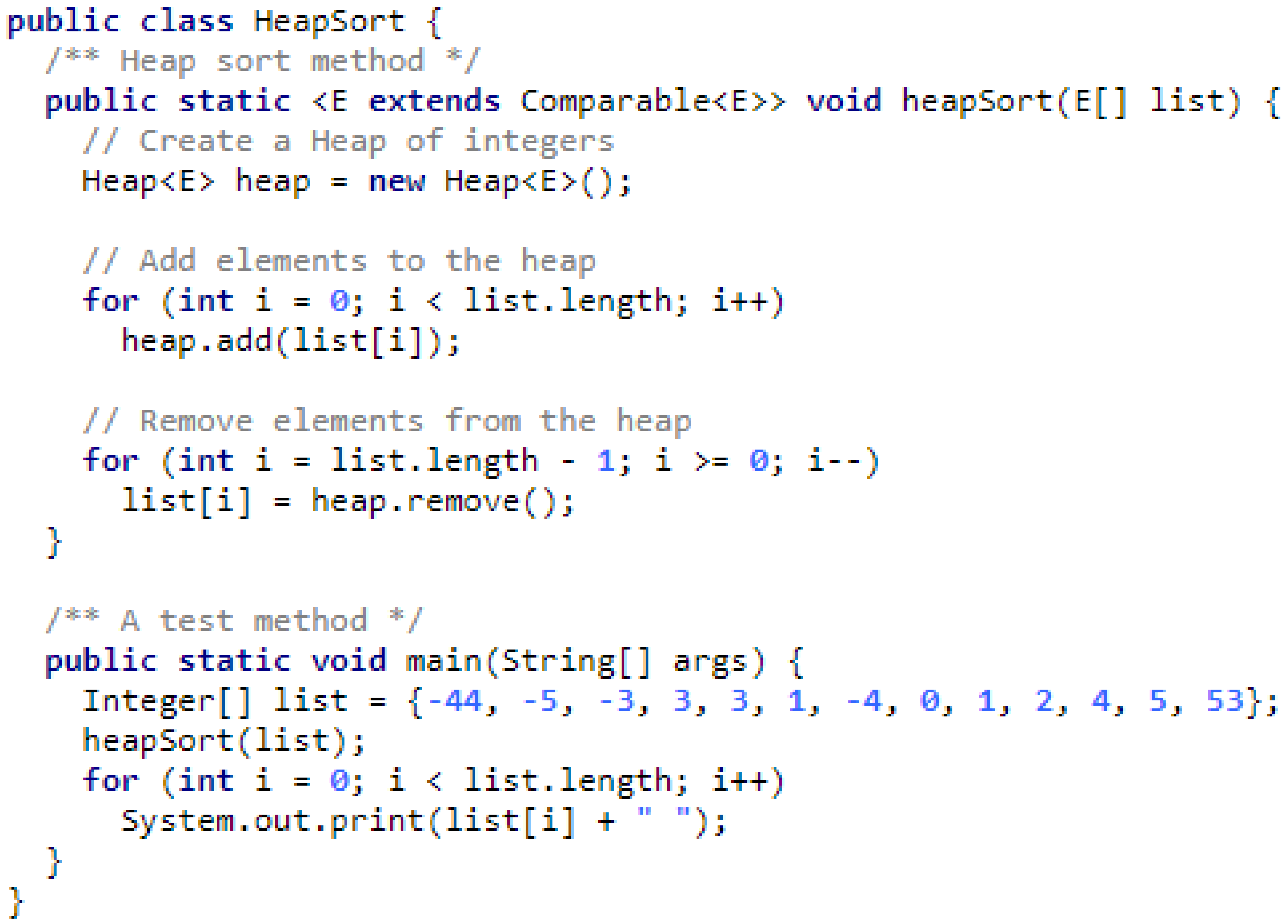
- findMax [or find-min]: find a maximum item of a max-heap, or a minimum item of a min-heap, respectively (a.k.a. peek)
- insert: adding a new key to the heap (a.k.a., push)
- ~~extractMax [or extract-min]: returns the node of maximum value from a max heap after removing it from the heap (a.k.a., pop)~~
- deleteMax [or delete-min]: removing the root node of a max heap (a.k.a., pop as seen in class)
- ~~createHeap~~: create an empty heap (the constructor)
- heapify: create a heap out of given array of elements
- size: return the number of items in the heap.
- capacity: return the number of items in the heap.
- isEmpty: return true if the heap is empty, false otherwise.

HEAPSORT ALGORITHM

- Removing a node from a heap always means removing the node with highest value.
  - This is easy to do because the maximum value is always in the root node.
- The problem is that once the root node is removed, the heap is incomplete and must be reorganized.
- There is an algorithm for making the heap complete again:
  - Remove the node at the root.
  - Move the node in the last position to the root.
  - Trickle the last node down until it is below.
- When this algorithm is applied continually, the data is removed from the heap in sorted order.

HEAPSORT

- source: http://www.cs.armstrong.edu/liang/intro9e/html/HeapSort.html



```java
public class HeapSort {
  /** Heap sort method */
  public static <E extends Comparable<E>> void heapSort(E[] list) {
    // Create a Heap of integers
    Heap<E> heap = new Heap<E>();

    // Add elements to the heap
    for (int i = 0; i < list.length; i++)
      heap.add(list[i]);

    // Remove elements from the heap
    for (int i = list.length - 1; i >= 0; i--)
      list[i] = heap.remove();
  }

  /** A test method */
  public static void main(String[] args) {
    Integer[] list = {-44, -5, -3, 3, 3, 1, -4, 0, 1, 2, 4, 5, 53};
    heapSort(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }
}
```

RUNNING TIME

- Discuss the running time for this algorithm …

EXTRA PROBLEMS

Write one C# program that includes all of the requirements below

- [75 points] Create a Heap class (use Heap ADT above!) that works with Strings.
- [15 points] Create a method that would sort (from largest to smallest) a given array using a Heap object. Use the prototype: public static void heapSort(string[] arr)
  - This method should be part of the same class where you Main() is, **not** part of the Heap class
- [10 points] Test your work in Main().
  - Create a new array, and sort it (from high to low) using heapsort. Display the array before and after sorting it.

- Make sure to include the big-Oh for each method you write (including Main())

EXTRA PROBLEMS

- [10 homework bonus points] Create a method that given an array checks whether or not the array represents a max heap.
- [5 homework bonus points] Create a method that given an array checks whether or not the array represents a min heap.
- [15 points] rewrite the Heap class so that instead of an array it uses an array list (use C# libraries for this). Make sure your code removes the size and count data members from the Heap class and instead uses the ones that come with the array list
- [15 points] rewrite the Heap class so that instead of using an array of int or string it uses an array objects of type **Student**. The class **Student** should contain a **name** and a **major**. The heap should store these objects based on student names (so student with the last alphabetically name should be stored at the top)

HOMEWORK FOR MODULE 6

None