CS32 Discussion
Week 9

Muhao Chen

muhaochen@ucla.edu

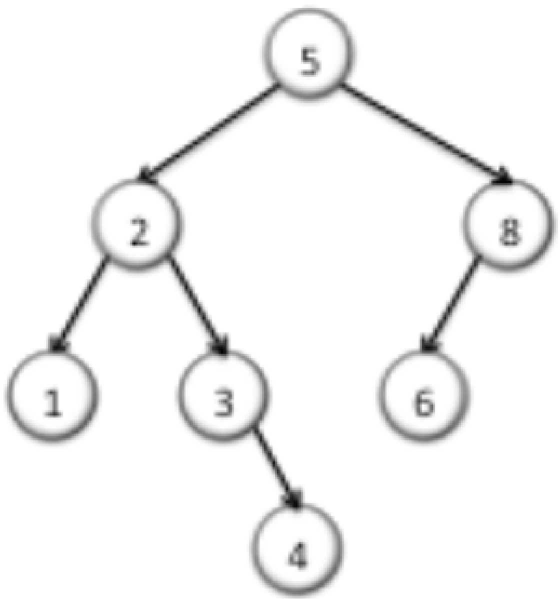
http://yellowstone.cs.ucla.edu/~muhao/

Outline

- Binary Search Tree
- Heap
- Hash Table

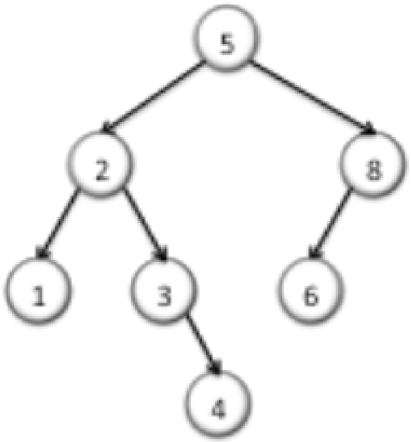
Binary Search Tree

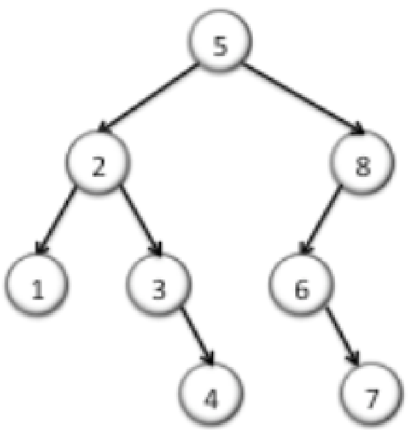
Binary Search Tree



- At all nodes:
  - All nodes in the left subtree have smaller values than the current node's value
  - All nodes in the right subtree have larger values than the current node's value
- Which traversal method should you use to:
  - print values in the increasing order?
  - print values in the decreasing order?

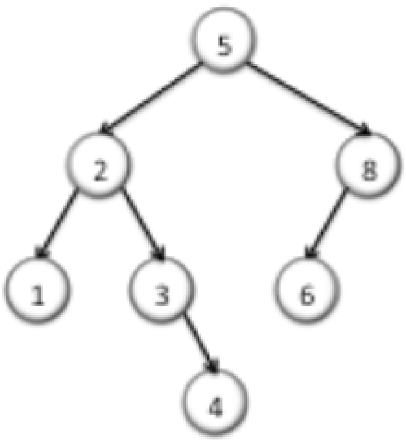
Insert

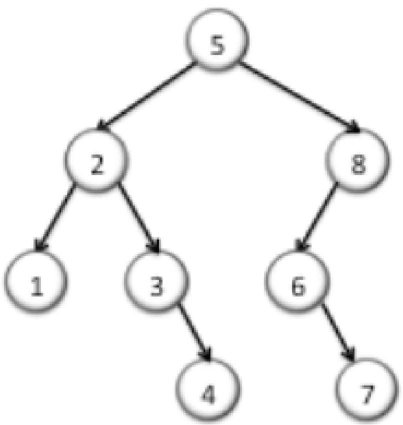





```
void insert(Node* &node, ItemType newVal)
{
    if (node == NULL)
    {
        node = new Node;
        node->val = newVal;
        node->left = node->right = NULL;
    }

    if (node->val > newVal)
        insert(node->left, newVal);
    else
        insert(node->right, newVal);
}
```

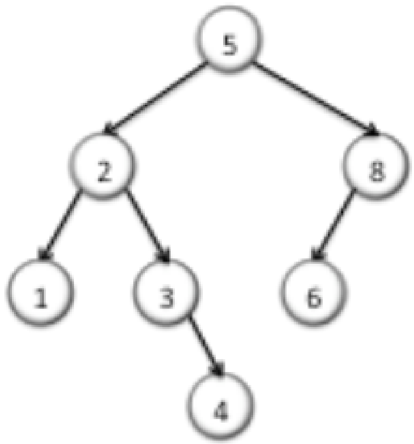
Insert





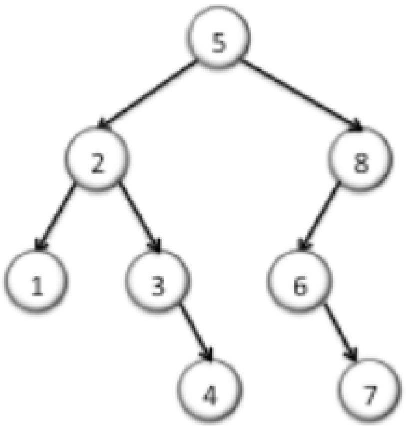
- **Average** time complexity?
  - as many steps as the height of the tree
  - full tree: $n = 2^{h+1} - 1 \approx 2^{h+1}$ nodes
  - $h \approx \log_2 n - 1$
  - Roughly, it takes O(log N).

Search

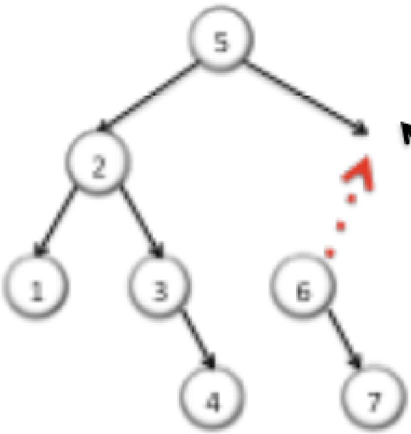



```
Node* search(const Node *node, ItemType value)
{
    if (node == NULL)
        return NULL;

    if (node->val == value)
        return node;
    else if (node->val > value)
        return search(node->left, value);
    else
        return search(node->right, value);
}
```

Removal

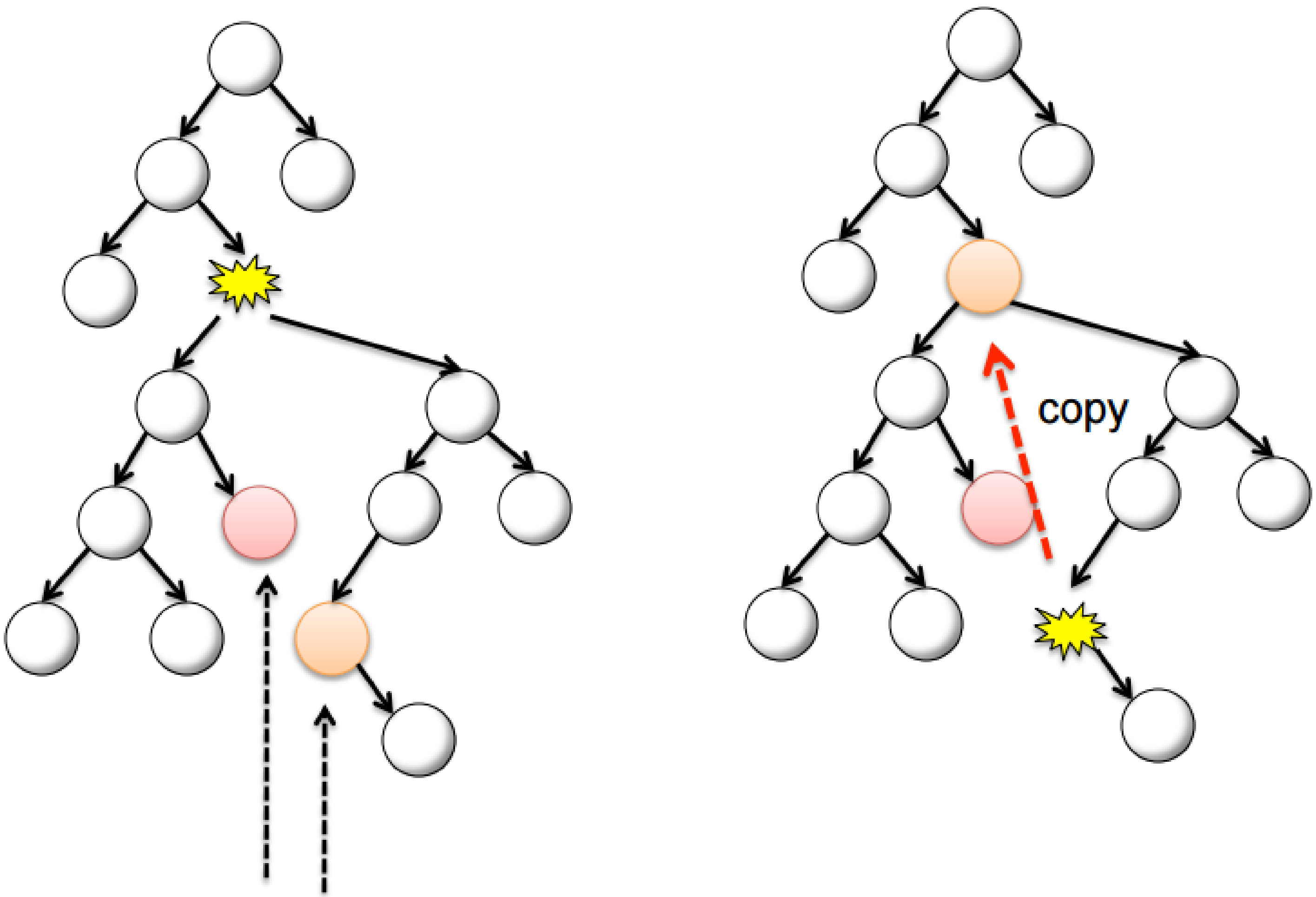


- A little tricky!
- General strategy:
  - Find a replacement.
  - Delete the node.
  - Replace.



- Case-by-case analysis
  - Case 1: the node is a leaf (easy)
  - Case 2: the node has one child
  - Case 3: the node has two children

Case 3



Use in-order traversal to identify these nodes

findMax

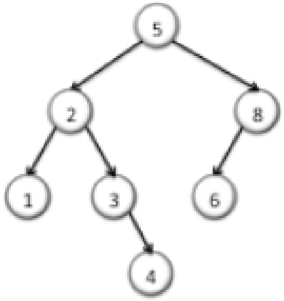
```
ItemType findMax(const Node *node)
{




}
```

FindMax

```
ItemType findMax(Node *node) {
    if (node -> right == NULL) return node->val;
    return findMax(node -> right);
}

//We assume the root is not NULL.
```

FindMin

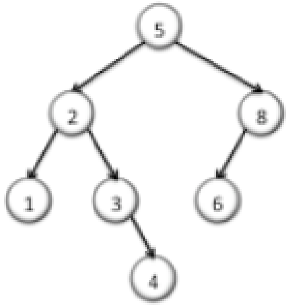
```
ItemType findMin(Node *node) {
    if (node -> left == NULL) return node->val;
    return findMax(node -> left);
}
```



valid

```
bool valid(const Node *node)
{



}
```

- At all nodes:
  - All nodes in the left subtree have smaller values than the current node's value
  - All nodes in the right subtree have larger values than the current node's value



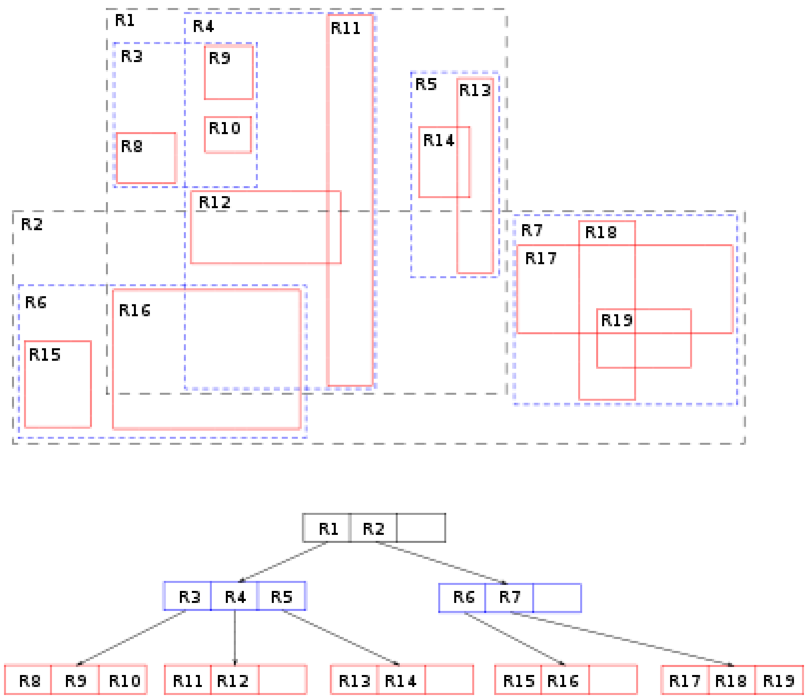
valid

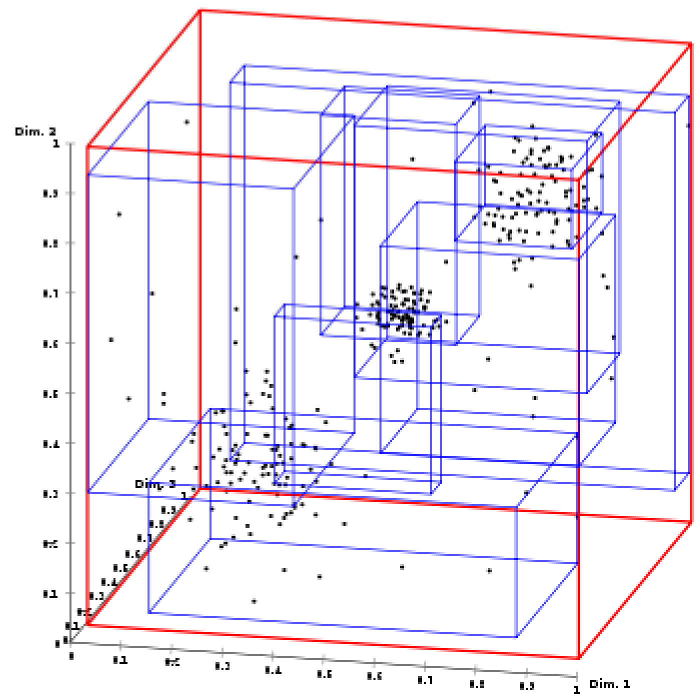
- At all nodes:
  - All nodes in the left subtree have smaller values than the current node' s value
  - All nodes in the right subtree have larger values than the current node' s value

```
bool valid(const Node *node)
{
    if (node == NULL)
        return true;

    if (node->left != NULL && findMax(node->left) > node->val)
        return false;

    if (node->right != NULL && findMin(node->right) < node->val)
        return false;

    return valid(node->left) && valid(node->right);
}
```

Other Representative Trees

- B+-Tree (CS143)
- R-Tree (Spatial Index Tree)
- Quad-tree

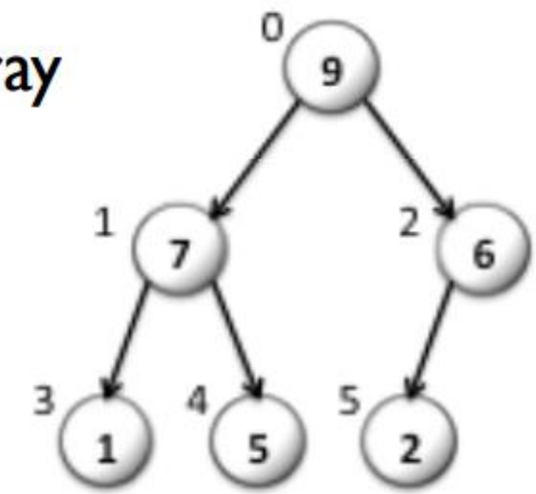




Heaps

Heaps

- A **heap** is a
  - complete binary tree
  - every node carries a value greater than or equal to its children's (maxHeap).
  - usually implemented as an array



9	7	6	1	5	2

parent = (i - 1) / 2

Left child: 2 * i + 1

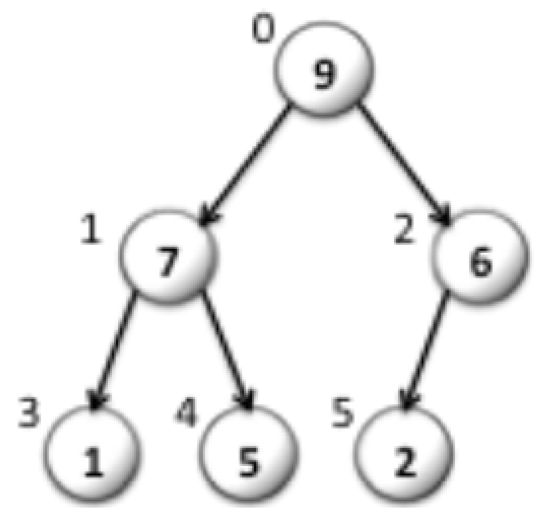
Right child: 2 * i + 2

Heaps: operations

- 3 operations for heaps
  - findMax (search)
  - insertNode (insert)
  - deleteMax (remove)

Body structure of a *Priority Queue*

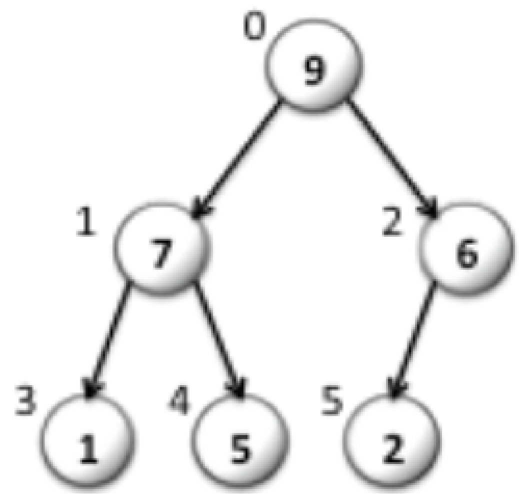
9	7	6	1	5	2



findMax

- What do you think?

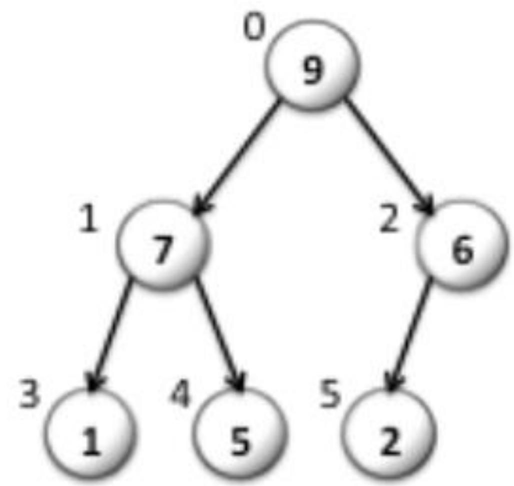
9	7	6	1	5	2



insertNode

- Not so trivial
- We first add the new node and fix it

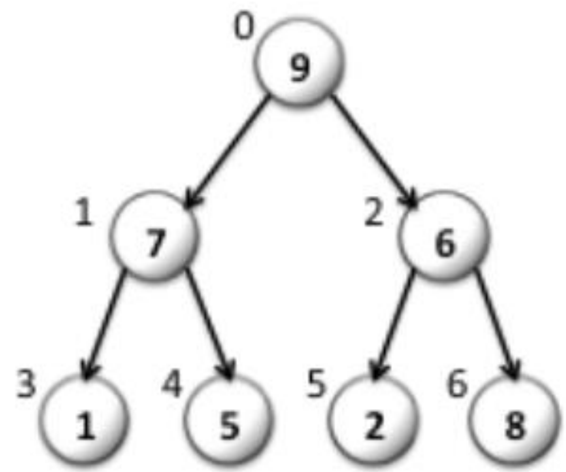
9	7	6	1	5	2



insertNode

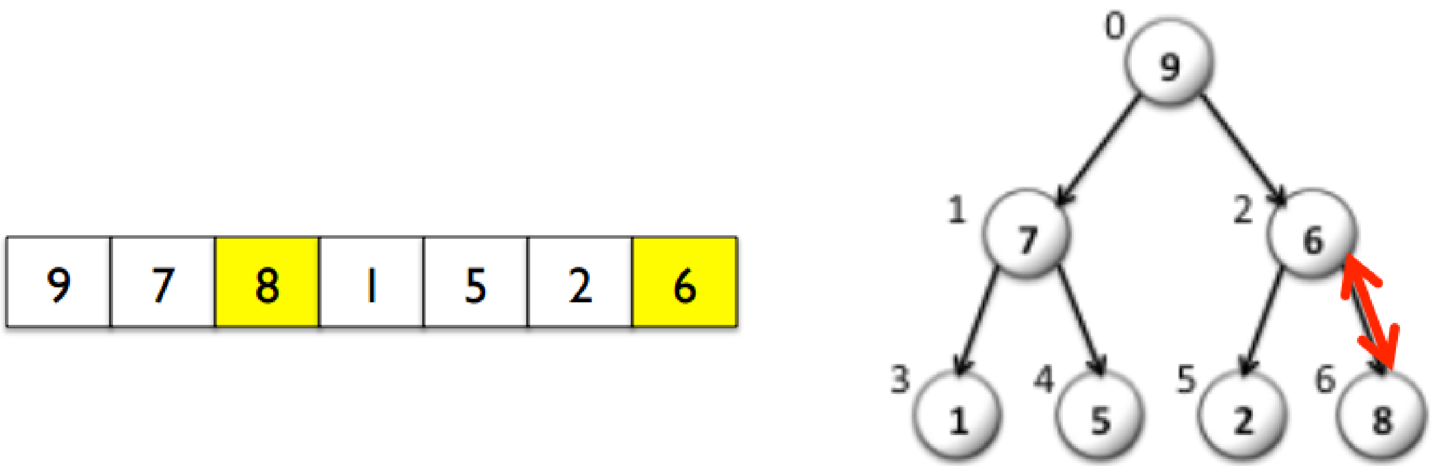
1. Add the new node to the tail.
2. Ask:
   - Is the new value greater than its parent?
   - If so: ??
   - Else: ??

9	7	6	1	5	2	8



insertNode

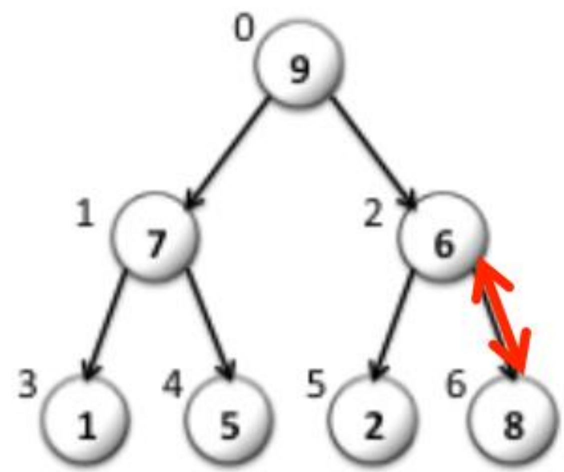
- What is the index of node i's parent in the array?



insertNode

- What is the index of node i's parent in the array?
  - parent = (i - 1) / 2

9	7	8	1	5	2	6

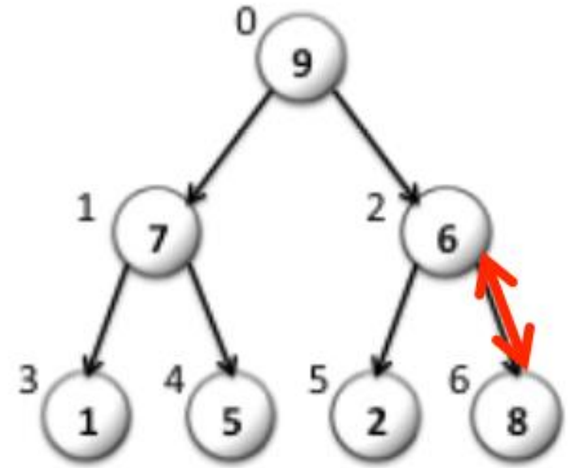


```
insertNode(int newVal, int heap[], int &size)
{
    heap[size] = newVal;    // assume enough space

    pos = size;

    parent = (pos – 1) / 2;

    while (parent >= 0 and heap[pos] > heap[parent])
    {
        swap(heap[pos], heap[parent]);
        pos = parent;

        parent = (pos – 1) / 2;
    }
    size++;
}
```

insertNode

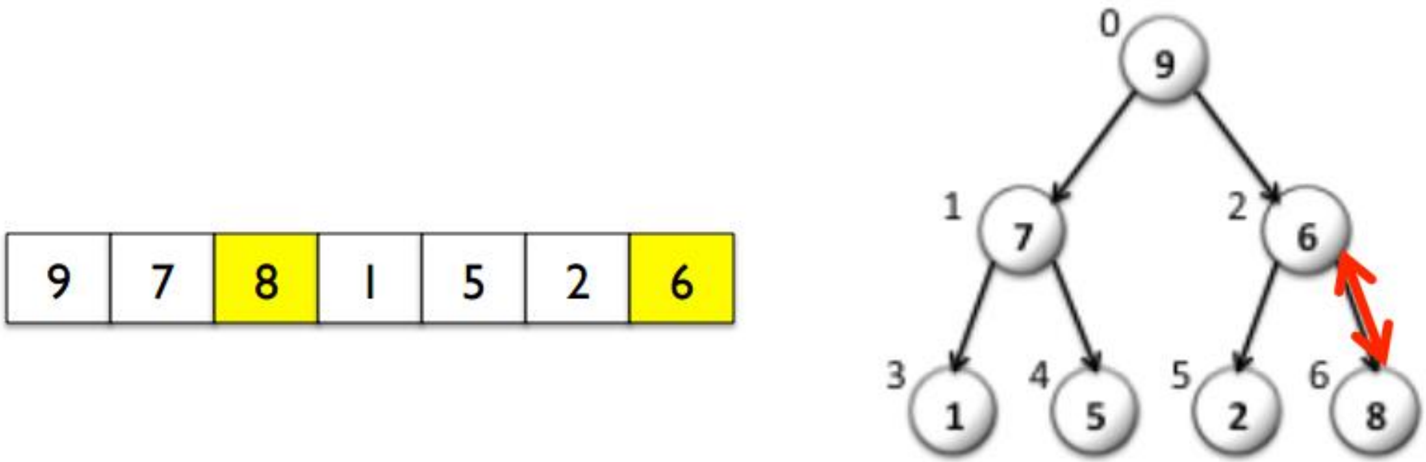
- Running time?

9	7	8	1	5	2	6



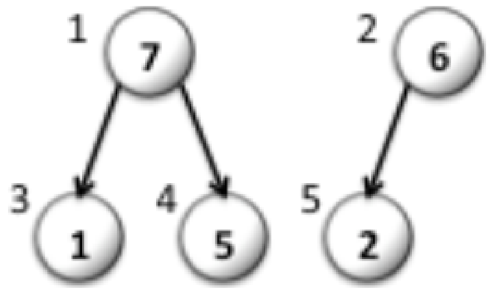
insertNode

- Running time?
  - proportional to the height of the tree: **O(log n)**

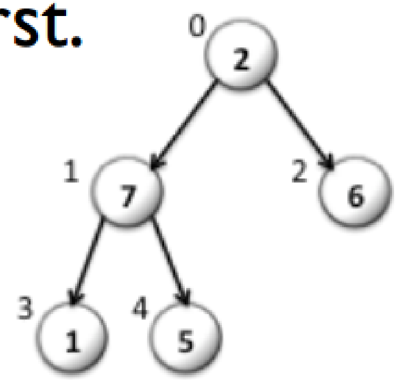


deleteMax

- Again, take the action first and fix it.



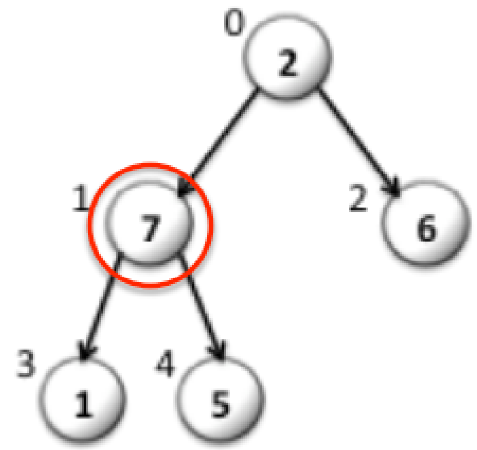
- Fill in the void first.



deleteMax

- Now compare the values of the two children, take the greater of the two (why?), and swap.
- What are the indices of
  - Left child:
  - Right child:

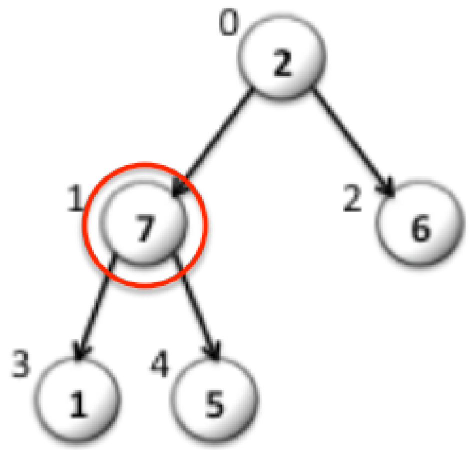
  of the node i?



deleteMax

- Now compare the values of the two children, take the greater of the two (why?), and swap.
- What are the indices of
  - Left child: 2 * i + 1
  - Right child: 2 * i + 2

  of the node i?



```
deleteMax(int heap[], int size)
{
    heap[0] = heap[size-1];
    size--;

    pos = 0;
    left_child = 1;

    while (left_child < size)     // if not a leaf
    {
        right_child = left_child + 1;

        // if right child exists
        if (right_child < size &&
            heap[right_child] > heap[left_child])
        {
            swap(heap[right_child], heap[pos]);
            pos = right_child;
        }
        // if only left child exists
        else if (heap[left_child] > heap[pos])
        {
            swap(heap[left_child], heap[pos]);
            pos = left_child;
        }
        else
            break;

        left_child =  pos * 2 + 1
    }
}
```

Cost of each operation of a max-heap

- findMax --- O(1)
- insert --- O(logn)
- deleteMax --- O(logn)

Heapsort

- Can you use a heap to sort a set of elements?

Heapsort

- Can you use a heap to sort a set of elements?
  - Insert all elements into a heap
  - Extract the maximum element from the heap one by one

Heapsort

- Can you use a heap to sort a set of elements?
  - Insert all elements into a heap
  - Extract the maximum element from the heap one by one
- Running time?

Heapsort

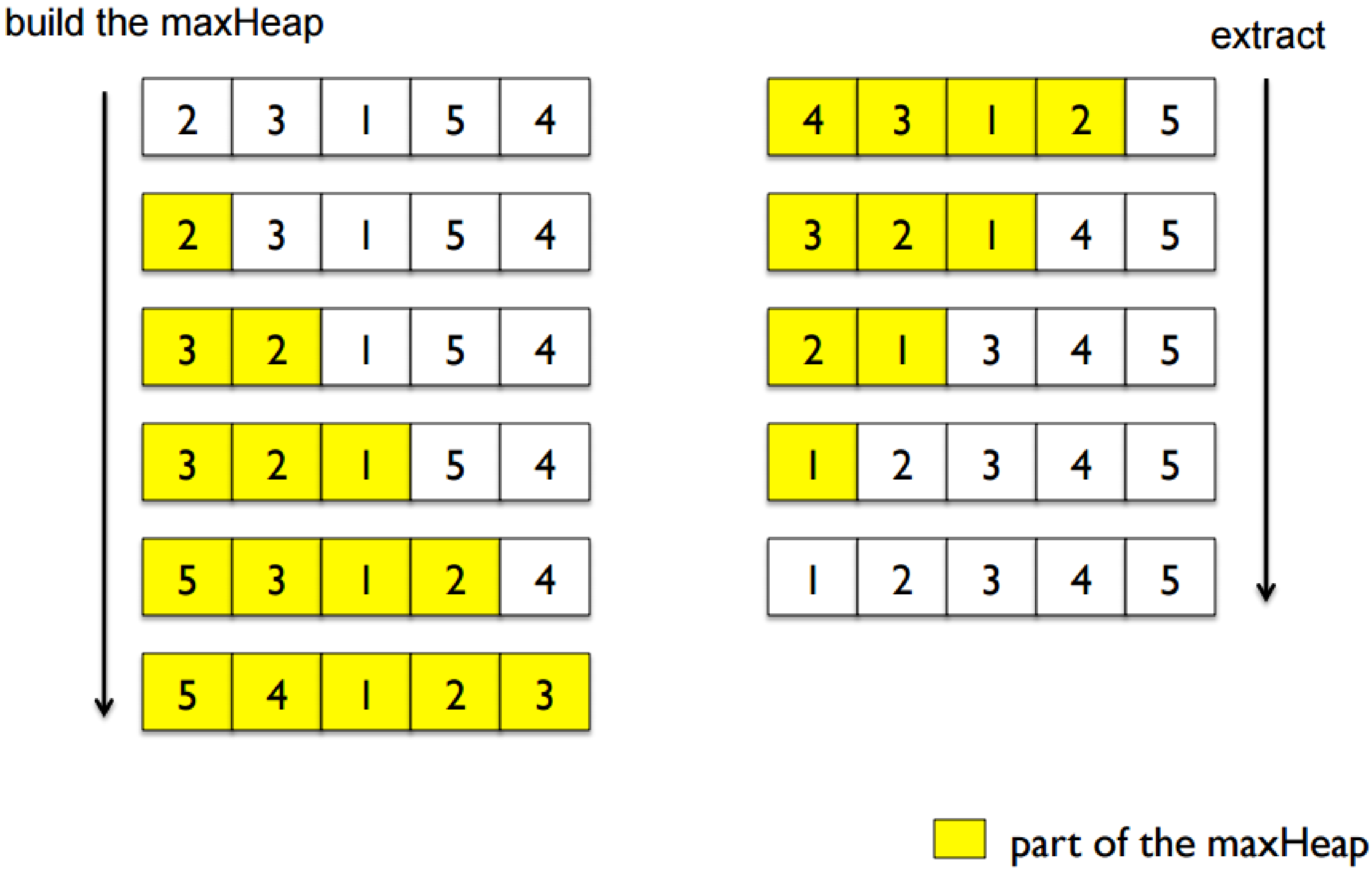
- Can you use a heap to sort a set of elements?
  - Insert all elements into a heap
  - Extract the maximum element from the heap one by one
- Running time?

  Inserting n items: n x O(log n) = O(n log n)

  Extracting n items: n x O(log n) = O(n log n)

  O(n log n) + O(n log n) = **O(n log n)**

In-place Heapsort



Question: top-*k* query

- How can we efficiently find *k* largest numbers from *n* numbers? (*n>>k, but k is not small)*
  - Sort? **Too costly!**
  - Scan (i.e. linear search) *k* times? **O(k*n) what if large *k?***

- Use heap (max-heap or min-heap?)
  - Min-heap!
  - Pop-out the *min* from heap and insert a number, if it's larger than *min*.
  - O(nlogk)

Question: merge *k* sorted linked lists

- If we have *k* **sorted** linked lists.
  - Each list has *n* nodes. (Thus totally *nk* nodes)
- What's the efficient way to merge them into one sorted list? (hint: O(*n***k*log*k*))

Question: merge *k* sorted linked lists

- Inefficient solution: Brute Force
- Keep linear searching the *k* heads and fetching the smallest until all lists are empty
- $O(nk^2)$

Question: merge *k* sorted linked lists

- Solution #1: Use minHeap.
  - Insert the head of each list to the heap.
  - Each time we pop-out a node from the heap and append it to the result list, insert the next node of that node from its list to the heap.
  - Do this until heap is empty.

- Complexity:
  - Each node takes *O(logk)* to be inserted in the heap, *O(logk)* to extract and *O(1)* to append to result.
  - *nk* nodes => *O(nklogk)*

Question: merge *k* sorted linked lists

- Solution #2: it is actually merge sort.
  - Merge each pair of sorted lists. *k* sorted lists become *k/2* sorted lists.
  - Again, merge each pair of sorted lists. *k/2* becomes *k*/4.
  - …
  - Do this until everything is merged into 1 list.

- Complexity:
  - Each stage of merge (e.g. from *k* lists to *k/2* lists) takes O(*nk)*. (*nk* times of comparison and append)
  - Altogether there're O(*logk*) stages of merge. => O(nklogk)

Hash-table

Hash Functions

- Hashing
  - Take a “key” and map it to a number

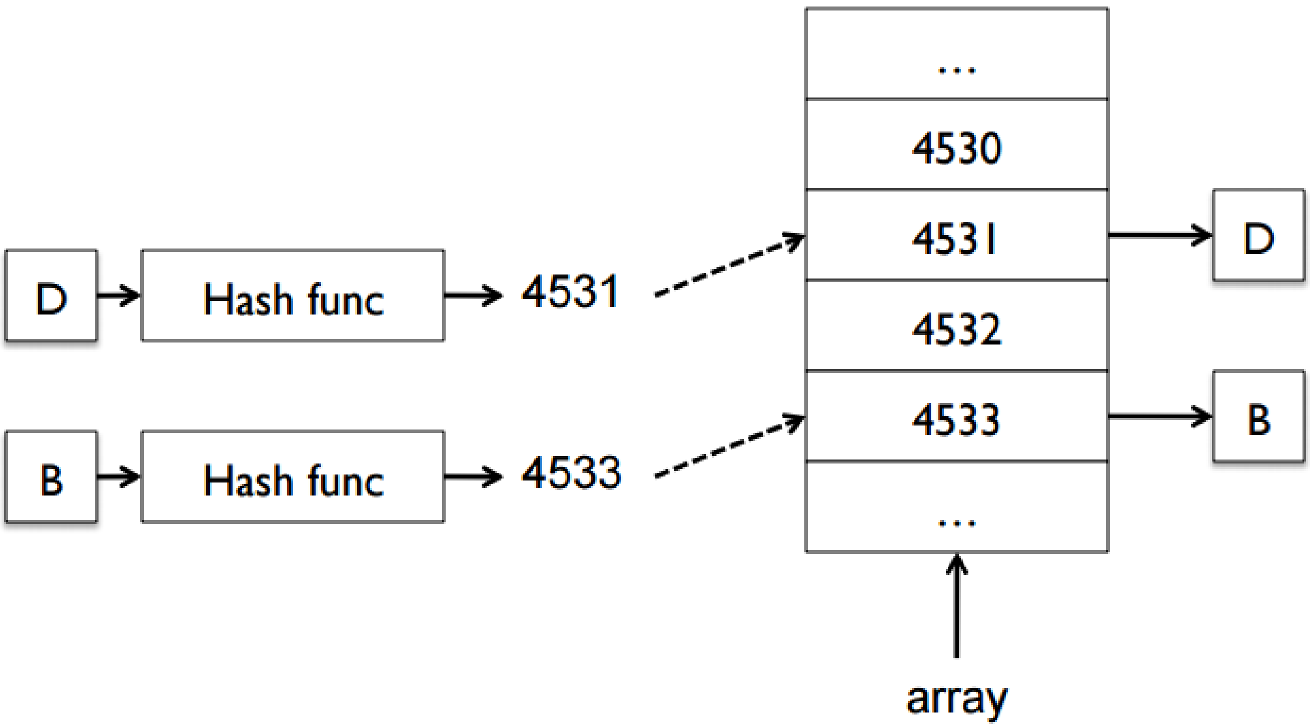
- A requirement for hash function H: should return the same value for the same key.
- A good hash function
  - spreads out the values: two different keys are likely to result in different hash values
  - computes each value quickly

FNV-1 Hash Function

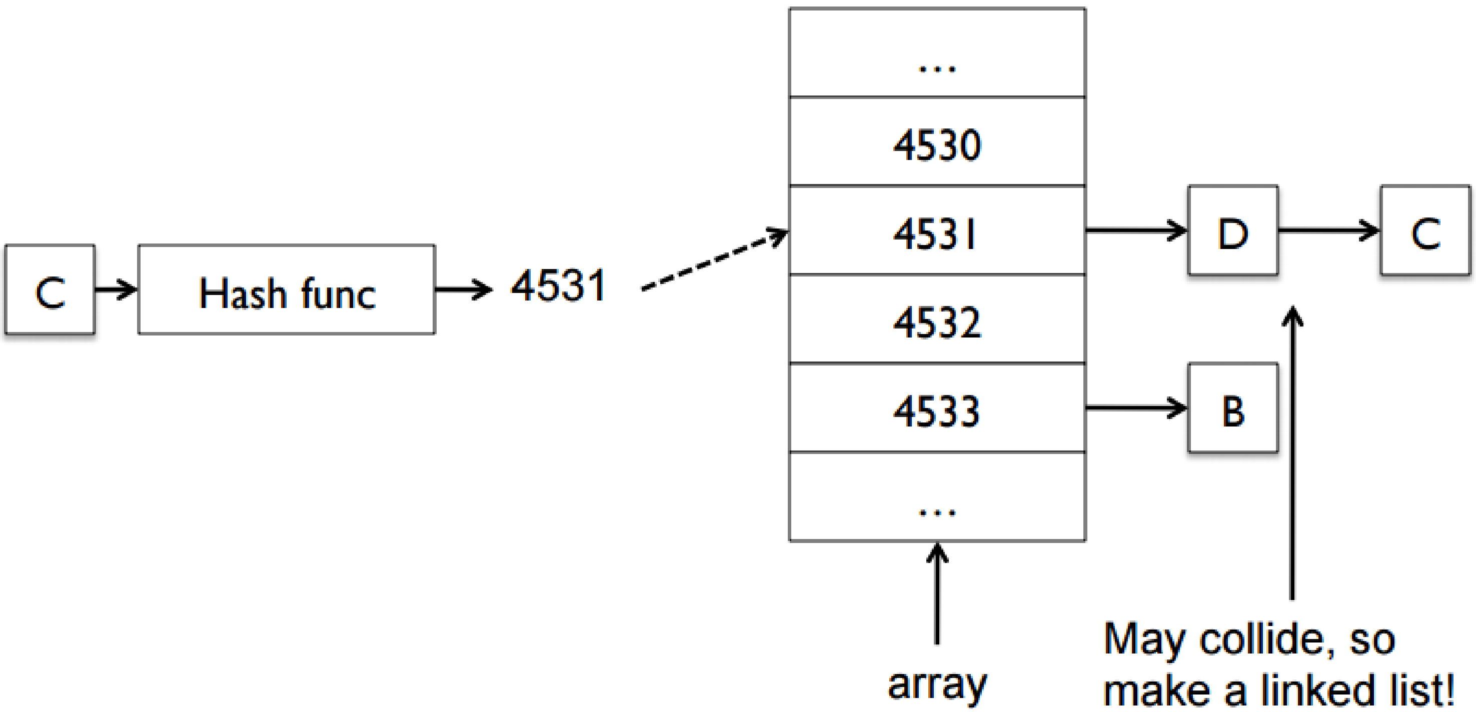
- A good hash function from string to int.

```
unsigned int FNV-1(string s) {
    unsigned int h = 2166136261U;
    for (int k = 0; k != s.size(); k++)
    {
        h += s[k];
        h *= 16777619;
    }
    return h;
}
```

Hash Table

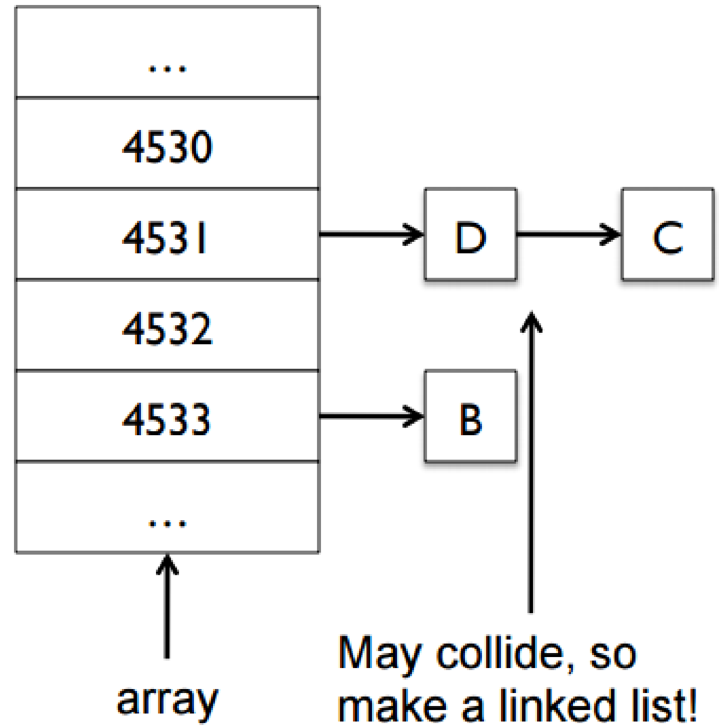


Hash Table


C
Hash func
4531
...
4530
4531
4532
4533
...
D
C
B
array
May collide, so
make a linked list!

Hash Table

- Running time
  - Insert?
  - Remove?
  - Search?

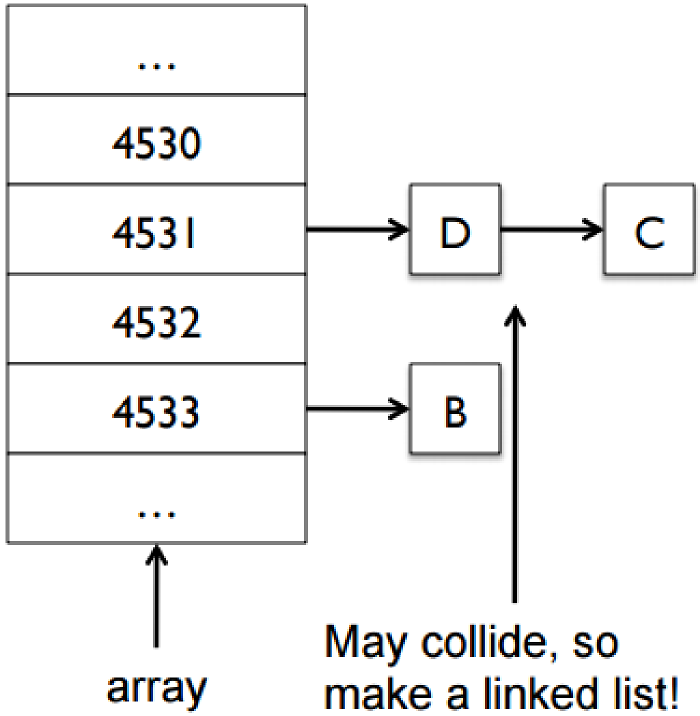


Hash Table

- **Closed hashing:**
  - Fixed number of buckets
  - All operations are O(n) with an extremely small constant of proportionality
    - (s.t. it can still beat a BST when *n* is large)
- **Open hashing:**
  - Load factor = #entries / #buckets
  - Changes hash function and makes available more buckets when *load factor* reaches certain *margin* (say, usually about 0.7)
  - O(1) for all operations

Hash Table

- Running time
  - Insert? **O(1)**
  - Remove? **O(1)**
  - Search? **O(1)**
- Looks great, but what are the limitations?



Question: Count top k frequent words in a document

- We have *n* words in a document, whose vocabulary size is *v* (i.e. *v* different words).
- The **most efficient** way to count the frequency for all words takes O(____) time complexity.
- After getting the frequency of each word, the **most efficient** way to get the top *k* frequent words takes O(____) time complexity.
- Totally the entire procedure takes O(____).

Question: Count top k frequent words in a document

- What is the efficient way?
- How do we record #occurrence for each word?
  - Hash table. *O(1)* to update a count when we see a word. Totally *O(n)*
  - Otherwise O(nlog*v*) if we use a tree. (*v*: size of vocabulary)
- How do we get the words with top-k frequency?
  - Again, min-heap + one pass scan. O(*v*log*k*)
- Totally *O*(*n* + *vlogk*)

Bugs in your software are actually special features :)