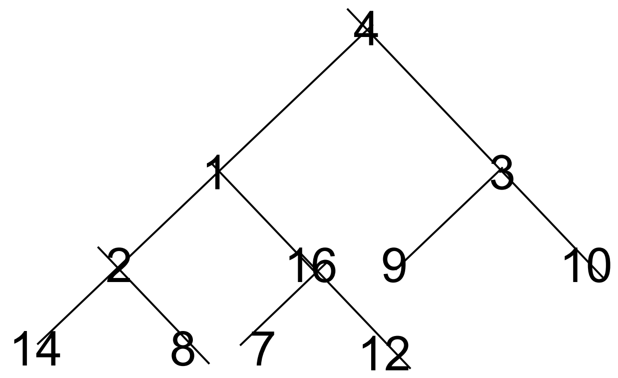
CSE408
Heap & Heap sort,Hashing

Lecture #18

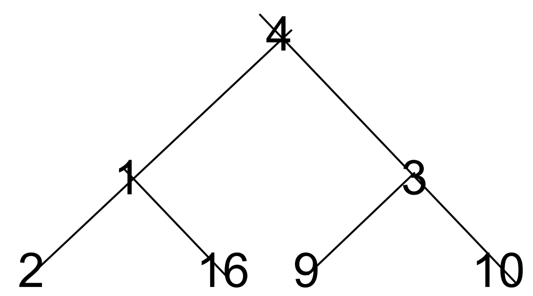
Special Types of Trees

- *Def:* **Full binary tree** = a binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

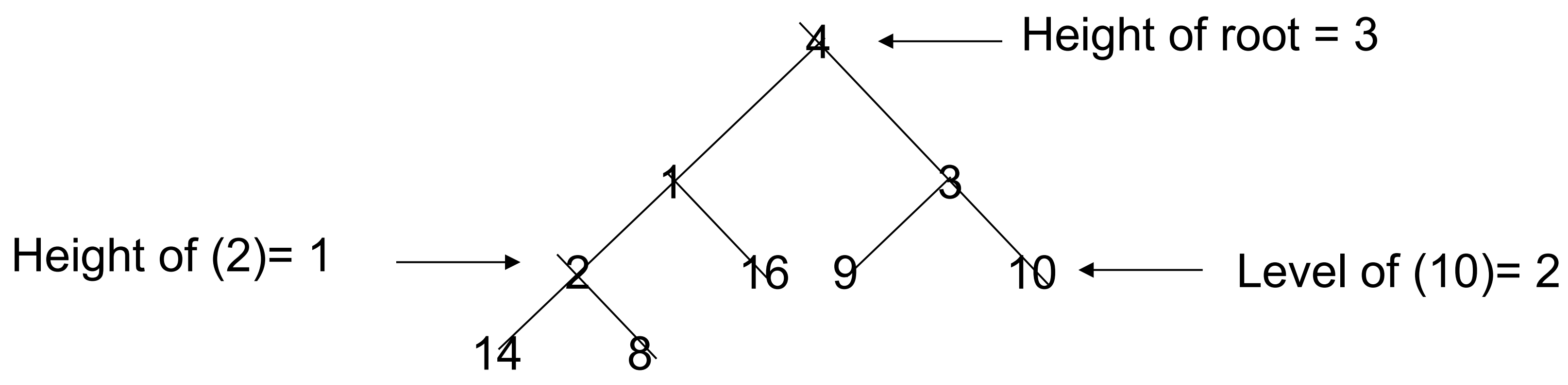
- *Def:* **Complete binary tree** = a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



Complete binary tree

Definitions

- **Height** of a node = the number of edges on the longest simple path from the node down to a leaf
- **Level** of a node = the length of a path from the root to the node
- **Height** of tree = height of root node

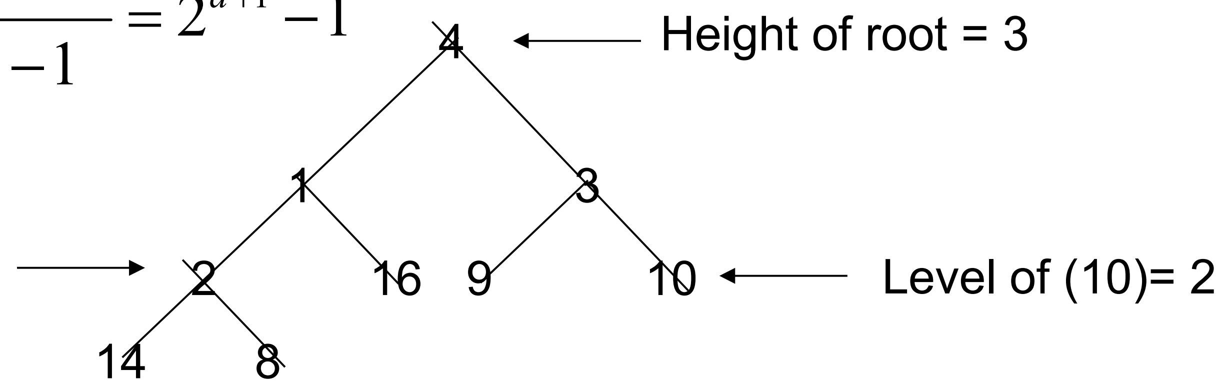


Useful Properties

- There are **at most** 2^l nodes at level (or depth) l of a binary tree
- A binary tree with depth height d has **at most** $2^{d+1} - 1$ nodes
- A binary tree with n nodes has depth height **at least** $\lfloor lgn \rfloor$

(see Ex 6.1-2, page 129)

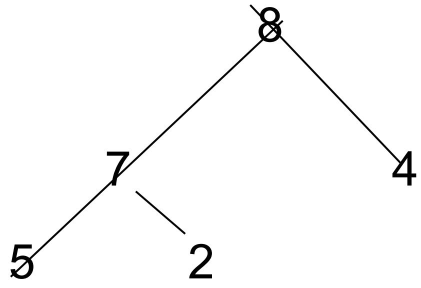
$$n \leq \sum_{l=0}^{d} 2^l = \frac{2^{d+1} - 1}{2 - 1} = 2^{d+1} - 1$$



The Heap Data Structure

- *Def:* A **heap** is a <u>nearly complete</u> binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x

$$\text{Parent}(x) \geq x$$



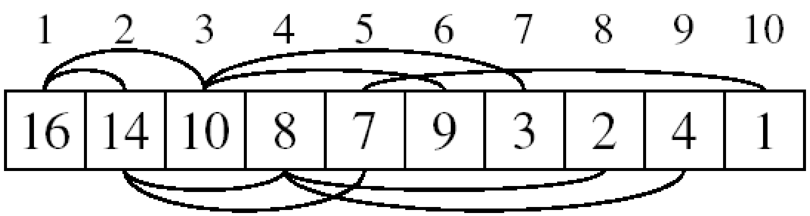
Heap

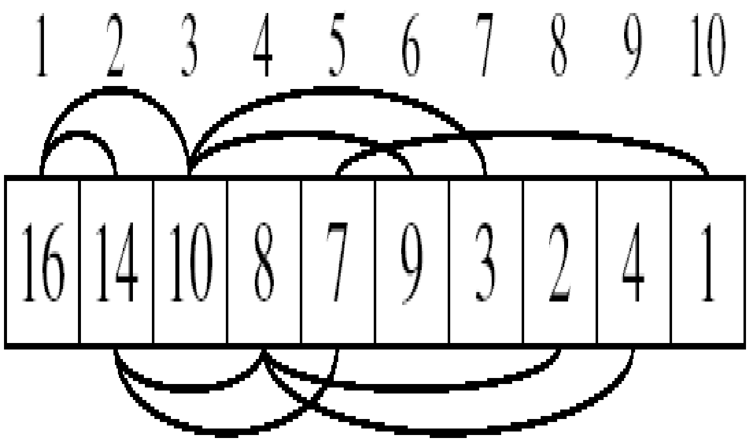
From the heap property, it follows that:
"The root is the maximum element of the heap!"

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves





Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

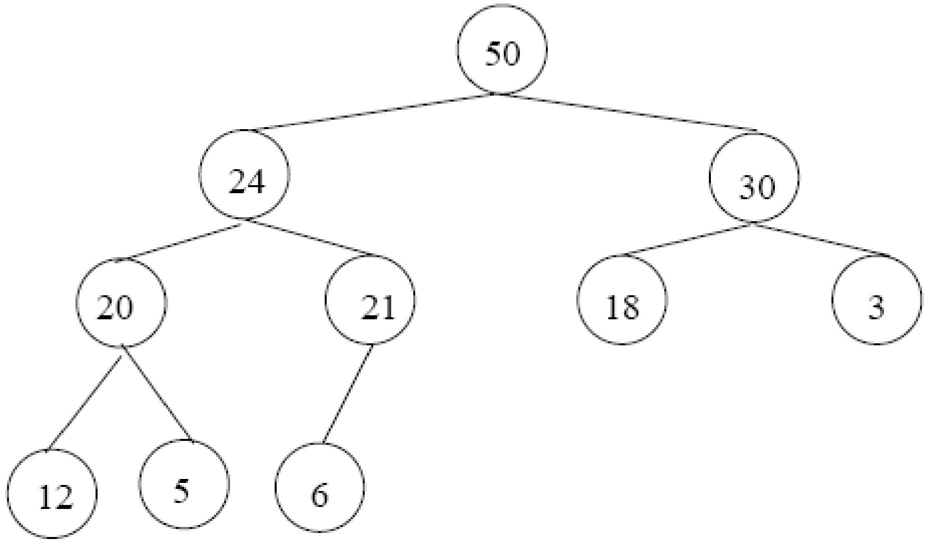
$$A[\mathrm{PARENT}(i)] \geq A[i]$$

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

$$A[\mathrm{PARENT}(i)] \leq A[i]$$

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

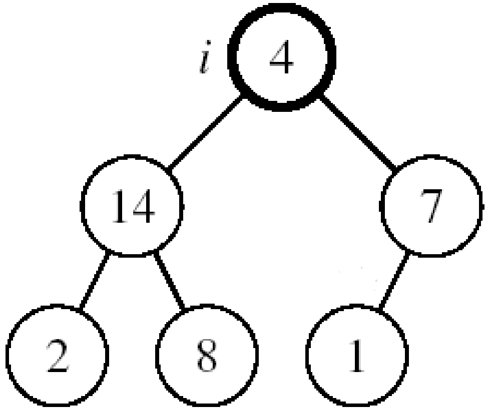


Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queues

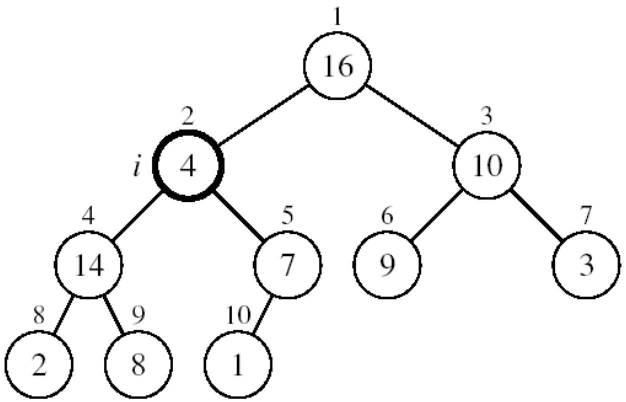
Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Left and Right subtrees of *i* are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children

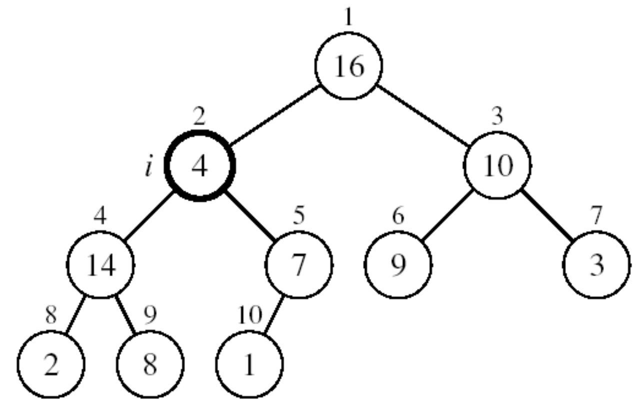


Example

MAX-HEAPIFY(A, 2, 10)



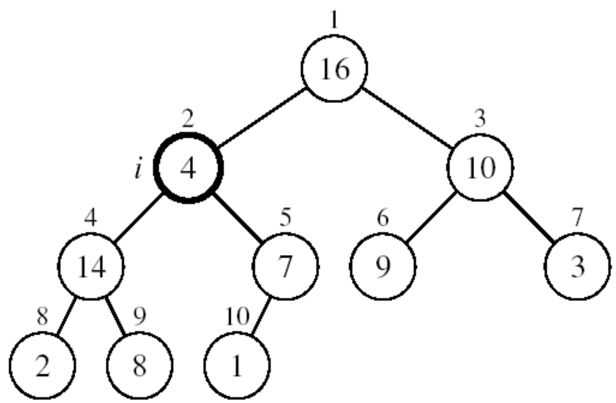
A[2] ↔ A[4]



A[2] violates the heap property

A[4] violates the heap property

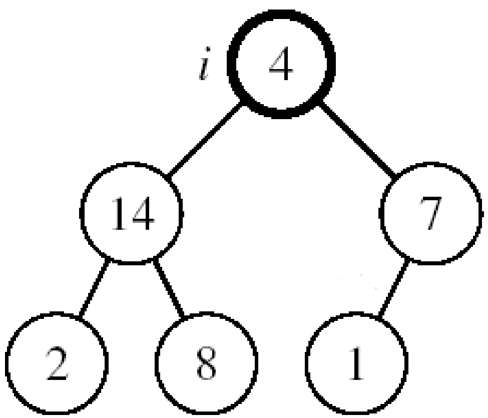
A[4] ↔ A[9]



Heap property restored

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - *A*[*i*] may be smaller than its children



Alg: MAX-HEAPIFY(*A*, *i*, *n*)

$l \leftarrow$ LEFT(i)
$r \leftarrow$ RIGHT(i)
if $l \leq n$ and $A[l] > A[i]$
    then *largest* $\leftarrow l$
    else *largest* $\leftarrow i$
if $r \leq n$ and $A[r] > A[largest]$
    then *largest* $\leftarrow r$
if *largest* $\neq i$
    then exchange $A[i] \leftrightarrow A[largest]$
        MAX-HEAPIFY(*A*, *largest*, *n*)

MAX-HEAPIFY Running Time

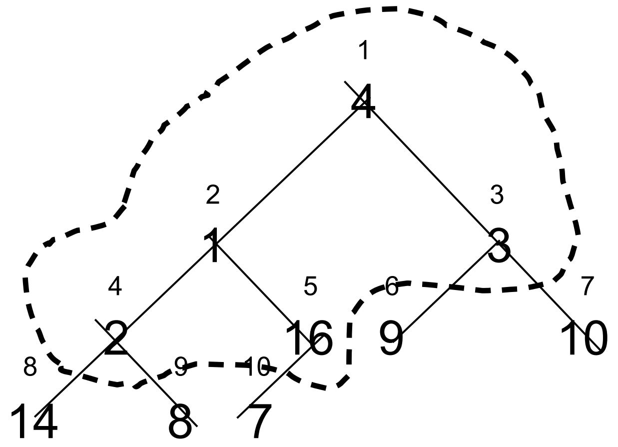
- Intuitively:
  - It traces a path from the root to a leaf (longest path length: h)
  - At each level, it makes exactly 2 comparisons
  - Total number of comparisons is $2h$
  - Running time is $O(h)$ or $O(lgn)$
- Running time of MAX-HEAPIFY is $O(lgn)$
- Can be written in terms of the height of the heap, as being $O(h)$
  - Since the height of the heap is $\lfloor lgn \rfloor$

Building a Heap

- Convert an array $A[1 \ldots n]$ into a max-heap ($n = length[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** i ← $\lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)

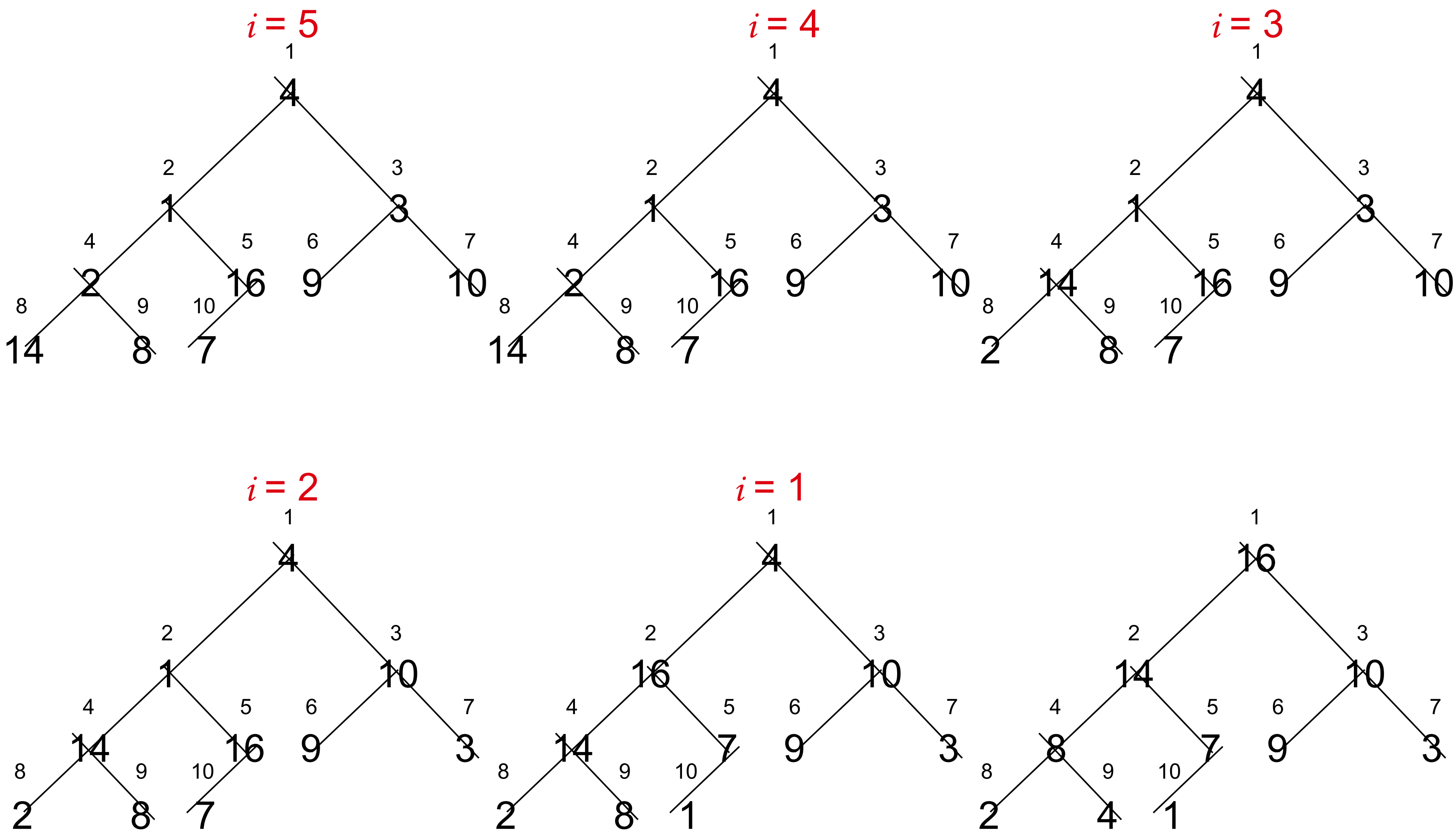


A:

4	1	3	2	16	9	10	14	8	7

Example: A

4	1	3	2	16	9	10	14	8	7



Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)

$O(\lg n)$ $O(n)$

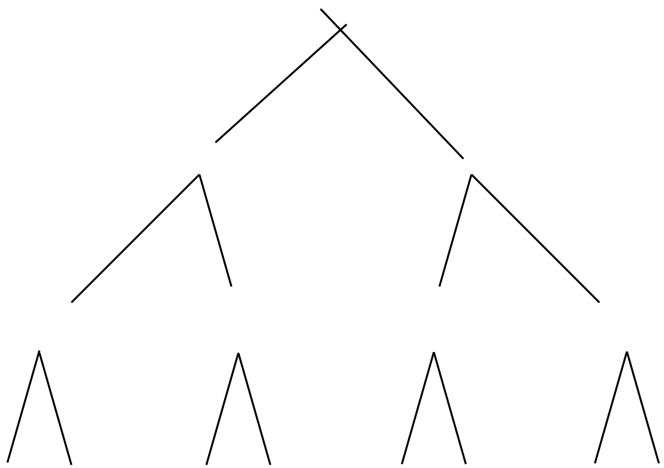
$\Rightarrow$ Running time: $O(n\lg n)$

- This is not an asymptotically tight upper bound

Running Time of BUILD MAX HEAP

- HEAPIFY takes $O(h)$ $\Rightarrow$ the cost of HEAPIFY on a node *i* is proportional to the height of the node *i* in the tree

$$= O(n) \quad = O(n) \quad = O(n)$$



Height	Level	No. of nodes
$h_0 = 3$ ($\lfloor \lg n \rfloor$)	$i = 0$	2^0
$h_1 = 2$	$i = 1$	2^1
$h_2 = 1$	$i = 2$	2^2
$h_3 = 0$	$i = 3$ ($\lfloor \lg n \rfloor$)	2^3

$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

Running Time of BUILD MAX HEAP

$$T(n)=\sum_{i=0}^{h} n_i h_i$$

Cost of HEAPIFY at level i $*$ number of nodes at that level

$$T(n)=\sum_{i=0}^{h} n_i h_i$$

Replace the values of n_i and h_i computed before

$$T(n)=\sum_{i=0}^{h} n_i h_i$$

Multiply by 2^h both at the nominator and denominator and write 2^i as $T(n)=\sum_{i=0}^{h} n_i h_i$

$$T(n)=\sum_{i=0}^{h} n_i h_i$$

Change variables: k = h - i

$$T(n)=\sum_{i=0}^{h} n_i h_i$$

The sum above is smaller than the sum of all elements to ∞ and h = lgn

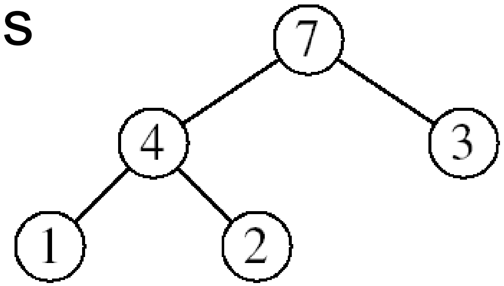
$$T(n)=\sum_{i=0}^{h} n_i h_i$$

The sum above is smaller than 2

Running time of BUILD-MAX-HEAP: $T(n) = O(n)$

Heapsort

- Goal:
  - Sort an array using heap representations
- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains



Example: A=[7, 4, 3, 1, 2]

A	1	2	3	4	7

MAX-HEAPIFY(A, 1, 4)

A	1	2	3	4	7

MAX-HEAPIFY(A, 1, 3)

A	1	2	3	4	7

MAX-HEAPIFY(A, 1, 2)

A	1	2	3	4	7

MAX-HEAPIFY(A, 1, 1)

A	1	2	3	4	7

A	1	2	3	4	7

Alg: HEAPSORT(A)

1. BUILD-MAX-HEAP(A) $O(n)$
2. **for** i ← length[A] **downto** 2
3.   **do** exchange $A[1] \leftrightarrow A[i]$ n-1 times
4.   MAX-HEAPIFY(A, 1, i - 1) $O(\lg n)$

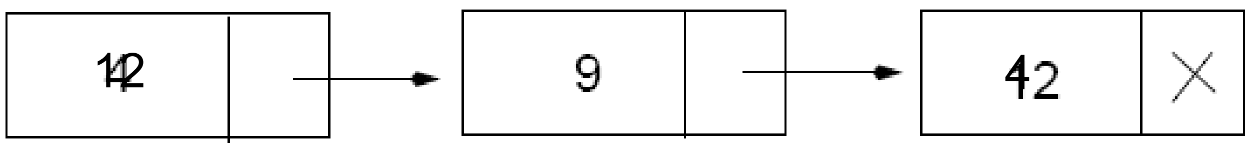
- Running time: $O(n\lg n)$ --- Can be shown to be $\Theta(n\lg n)$

Priority Queues

Properties

- Each element is associated with a value (priority)

- The key with the highest (or lowest) priority is extracted first



Operations on Priority Queues

- Max-priority queues support the following operations:
  - INSERT(S, x): inserts element x into set S
  - EXTRACT-MAX(S): removes and returns element of S with largest key
  - MAXIMUM(S): returns element of S with largest key
  - INCREASE-KEY(S, x, k): increases value of element x's key to k (Assume $k \geq x$'s current key value)

HEAP-MAXIMUM

Goal:

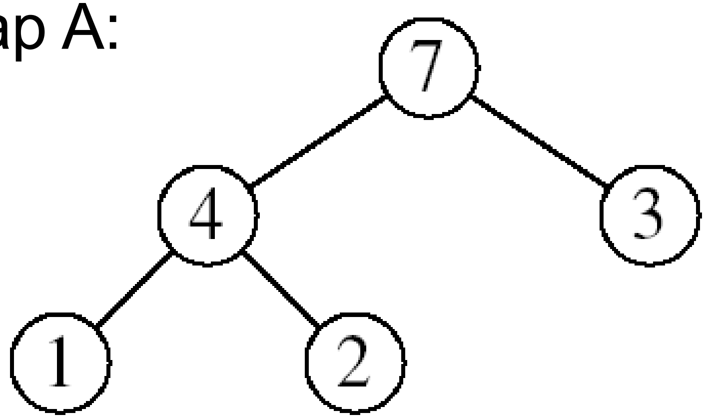
- Return the largest element of the heap

Alg: HEAP-MAXIMUM(*A*)

Running time: $O(1)$

1. **return** $A[1]$

Heap A:



Heap-Maximum(A) returns 7

HEAP-EXTRACT-MAX

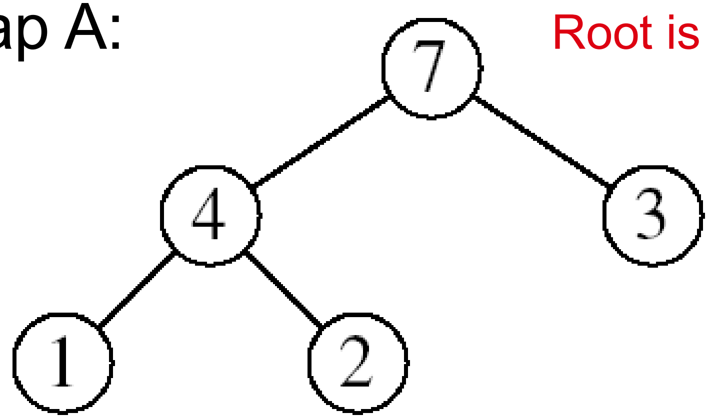
Goal:

- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

Idea:

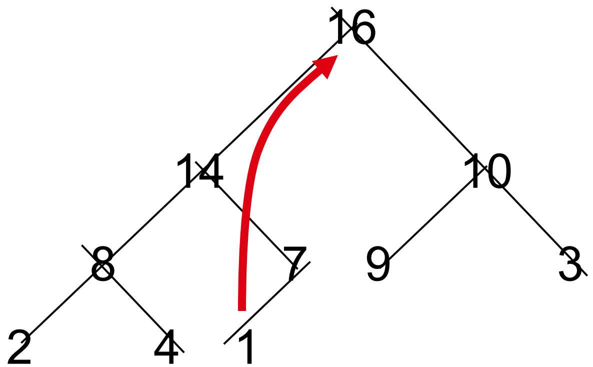
- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1

Heap A:

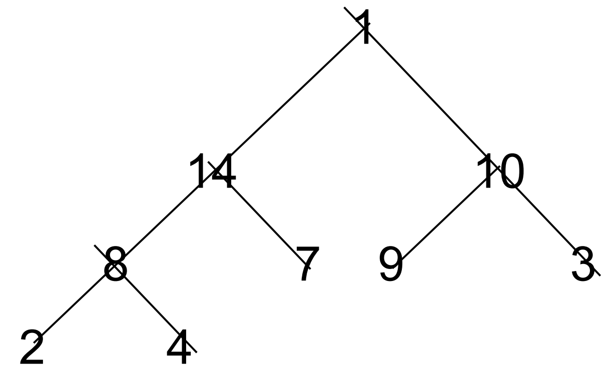


Root is the largest element

Example: HEAP-EXTRACT-MAX

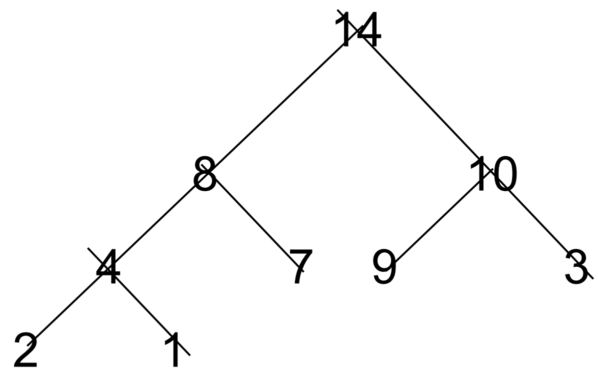


max = 16



Heap size decreased with 1

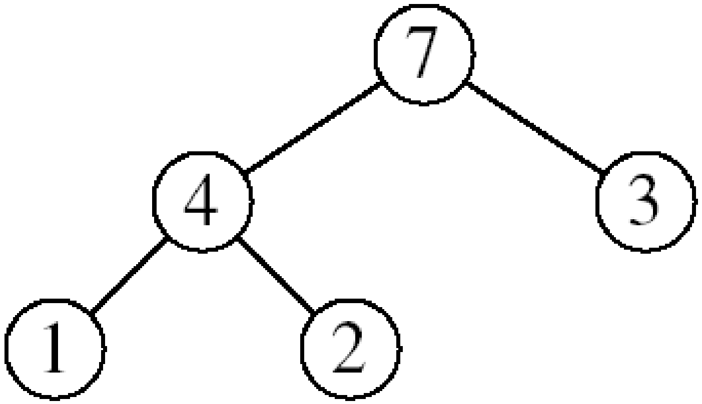
Call MAX-HEAPIFY(A, 1, n-1)



HEAP-EXTRACT-MAX

Alg: HEAP-EXTRACT-MAX(A, n)

1. **if** $n < 1$
2. **then error** "heap underflow"
3. $max \leftarrow A[1]$
4. $A[1] \leftarrow A[n]$
5. MAX-HEAPIFY(A, 1, n-1)
6. **return** max



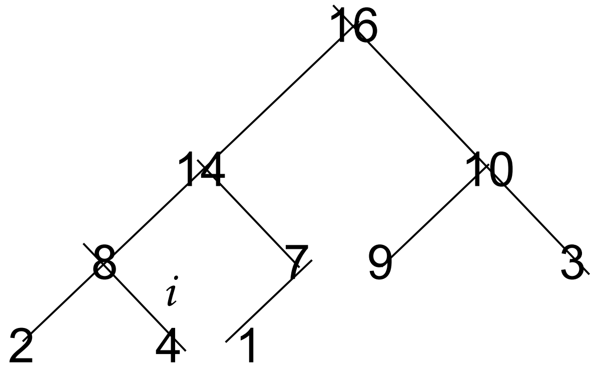
remakes heap

Running time: $O(\lg n)$

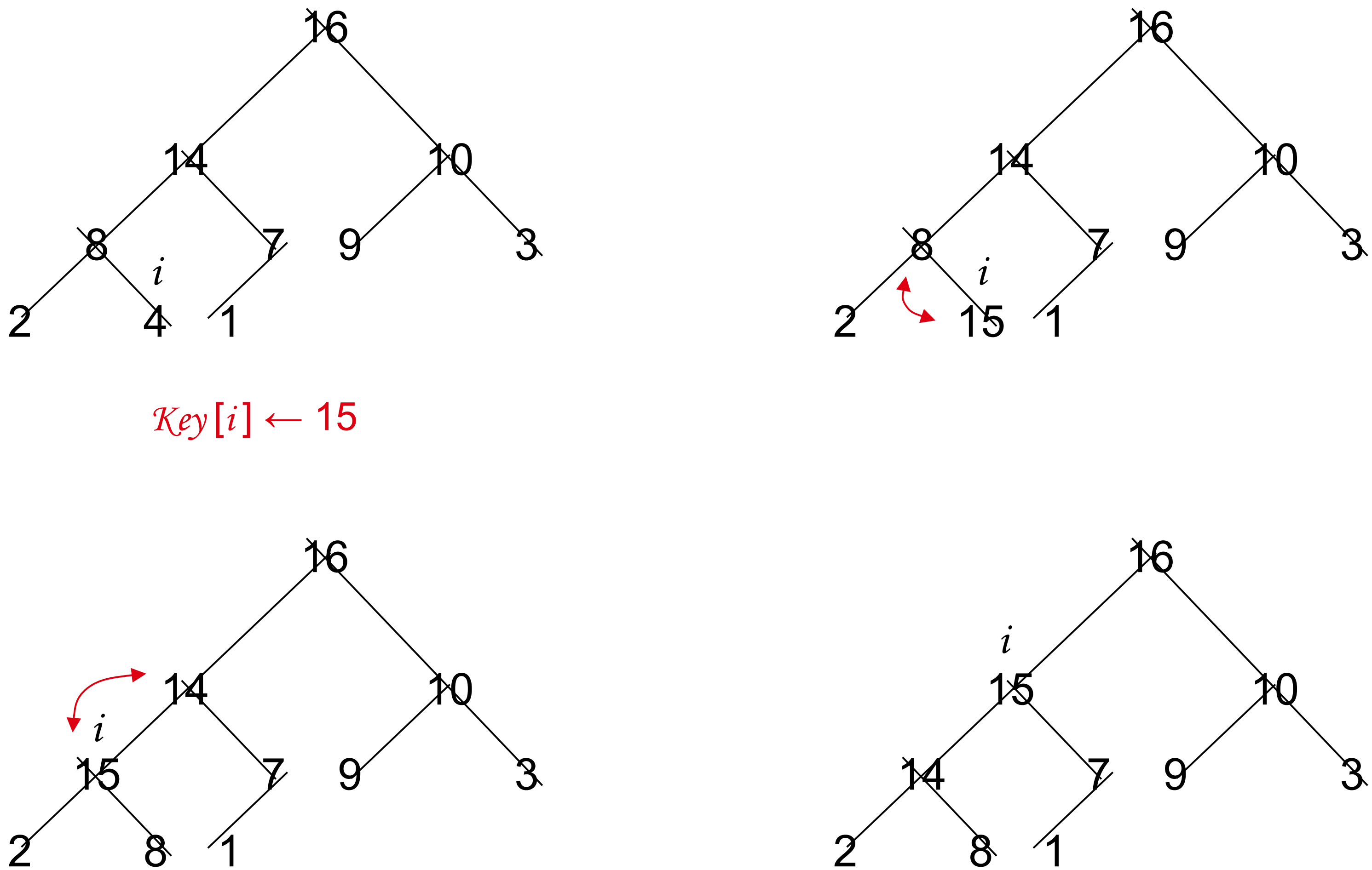
HEAP-INCREASE-KEY

- Goal:
  - Increases the key of an element i in the heap
- Idea:
  - Increment the key of $A[i]$ to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key

Key [i] $\leftarrow$ 15



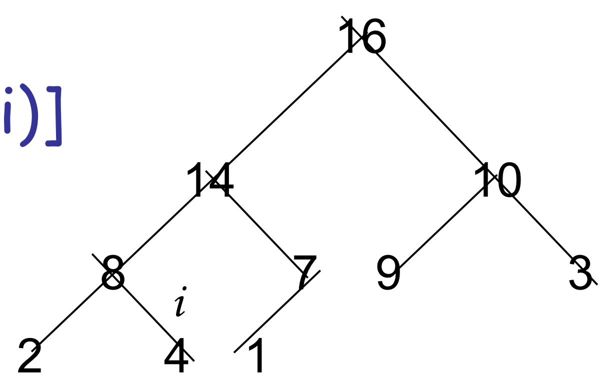
Example: HEAP-INCREASE-KEY



HEAP-INCREASE-KEY

Alg: HEAP-INCREASE-KEY(*A*, i, key)

1. **if** key < *A*[i]
2. **then error** "new key is smaller than current key"
3. *A*[i] ← key
4. **while** i > 1 and *A*[PARENT(i)] < *A*[i]
5. **do** exchange *A*[i] ↔ *A*[PARENT(i)]
6. i ← PARENT(i)

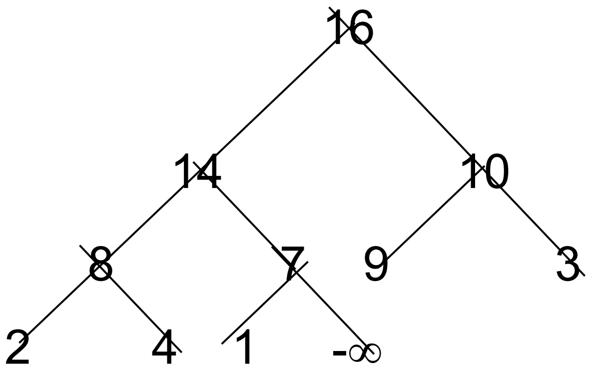


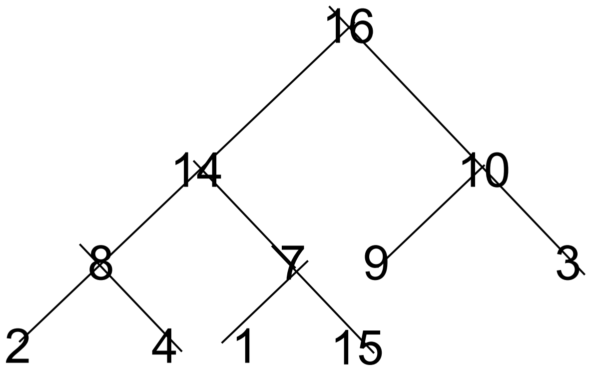
Key [i] ← 15

- Running time: *O*(lgn)

MAX-HEAP-INSERT

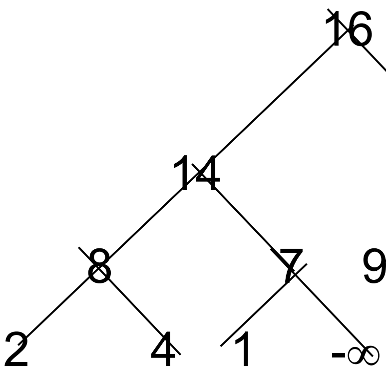
- Goal:
  - Inserts a new element into a max-heap
- Idea:
  - Expand the max-heap with a new element whose key is -∞
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property



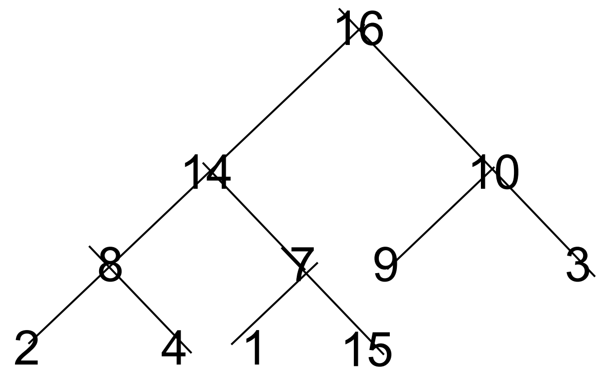


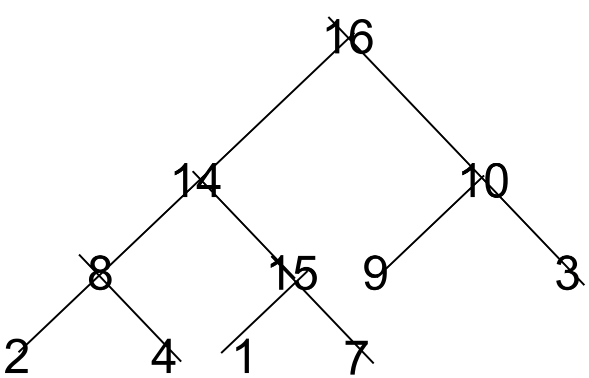
Example: MAX-HEAP-INSERT

Insert value 15:
- Start by inserting -∞

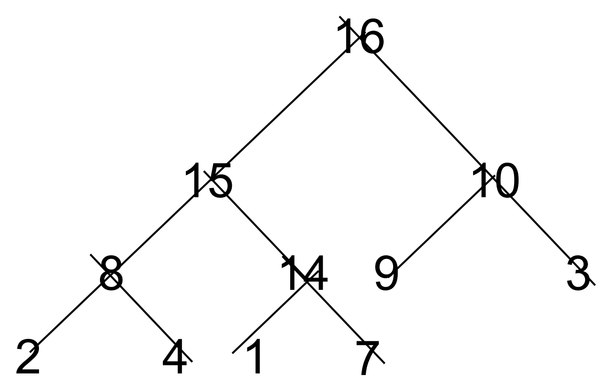


Increase the key to 15
Call HEAP-INCREASE-KEY on A[11] = 15





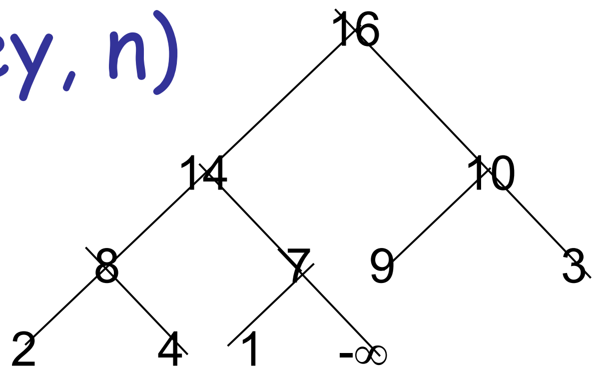
The restored heap containing
the newly added element



MAX-HEAP-INSERT

Alg: MAX-HEAP-INSERT(A, key, n)

1. heap-size[A] $\leftarrow n + 1$
2. $A[n + 1] \leftarrow -\infty$
3. HEAP-INCREASE-KEY(A, $n + 1$, key)



Running time: $O(\lg n)$

Summary

- We can perform the following operations on heaps:
  - MAX-HEAPIFY $O(\lg n)$
  - BUILD-MAX-HEAP $O(n)$
  - HEAP-SORT $O(n\lg n)$
  - MAX-HEAP-INSERT $O(\lg n)$
  - HEAP-EXTRACT-MAX $O(\lg n)$
  - HEAP-INCREASE-KEY $O(\lg n)$
  - HEAP-MAXIMUM $O(1)$

Average $O(\lg n)$

Hash Functions

- If the input keys are integers then simply *Key* mod *TableSize* is a general strategy.
  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are strings, hash function needs more care.
  - First convert it into a numeric value.

Some methods

- **Truncation:**
  - e.g. 123456789 map to a table of 1000 addresses by picking 3 digits of the key.
- **Folding:**
  - e.g. 123|456|789: add them and take mod.
- **Key mod N:**
  - N is the size of the table, better if it is prime.
- **Squaring:**
  - Square the key and then truncate
- **Radix conversion:**
  - e.g. 1 2 3 4 treat it to be base 11, truncate if necessary.

Hash Function 1

- Add up the ASCII values of all characters of the key.

```
int hash(const string &key, int tableSize)
{
    int hasVal = 0;

    for (int i = 0; i < key.length(); i++)
        hashVal += key[i];
    return hashVal % tableSize;
}
```

- Simple to implement and fast.
- However, if the table size is large, the function does not distribute the keys well.
  - e.g. Table size =10000, key length <= 8, the hash function can assume values only between 0 and 1016

Hash Function 2

- Examine only the first 3 characters of the key.

```
int hash (const string &key, int tableSize)
{
   return (key[0]+27 * key[1] + 729*key[2]) % tableSize;
}
```

- In theory, **26** * **26** * **26** = **17576** different words can be generated. However, English is not random, only **2851** different combinations are possible.
- Thus, this function although easily computable, is also not appropriate if the hash table is reasonably large.

Hash Function 3

$$hash(key) = \sum_{i=0}^{KeySize-1} Key[KeySize - i - 1] \cdot 37^{i}$$

```
int hash (const string &key, int tableSize)
{
   int hashVal = 0;

   for (int i = 0; i < key.length(); i++)
    hashVal = 37 * hashVal + key[i];

   hashVal %=tableSize;
   if (hashVal < 0)    /* in case overflows occurs */
    hashVal += tableSize;

   return hashVal;
};
```

Hash function for strings:

98 108 105 → key[i]

key a l i

0 1 2 → i

KeySize = 3;

hash(“ali”) = (105 * 1 + 108*37 + 98*37^2) % 10,007 = 8172

“ali” → hash function → **ali**

0
1
2
……
ali 8172
……
10,006 (TableSize)

Double Hashing

- A second hash function is used to drive the collision resolution.
  - $f(i) = i * hash_2(x)$
- We apply a second hash function to x and probe at a distance $hash_2(x)$, $2*hash_2(x)$, … and so on.
- The function $hash_2(x)$ must never evaluate to zero.
  - e.g. Let $hash_2(x) = x \bmod 9$ and try to insert 99 in the previous example.
- A function such as $hash_2(x) = R – (x \bmod R)$ with R a prime smaller than TableSize will work well.
  - e.g. try R = 7 for the previous example.(7 - x mode 7)

Hashing Applications

- Compilers use hash tables to implement the *symbol table* (a data structure to keep track of declared variables).
- Game programs use hash tables to keep track of positions it has encountered (*transposition table*)
- Online spelling checkers.

Summary

- Hash tables can be used to implement the insert and find operations in constant average time.
  - it depends on the load factor not on the number of items in the table.
- It is important to have a prime TableSize and a correct choice of load factor and hash function.
- For separate chaining the load factor should be close to 1.
- For open addressing load factor should not exceed 0.5 unless this is completely unavoidable.
  - Rehashing can be implemented to grow (or shrink) the table.