Sets

Terminology. Operations. Set-Based ADTs. Implementations. ADT Dictionary. Direct Access Tables. Hash Tables. Mapping ADT. Priority Queue ADT. Partially Ordered Trees. Heaps.

Set terminology

- <u>Set</u> : well-defined collection of distinct objects.
- An element of a set is any object in the set.
  - ∈ - "belongs to" or "is an element of"
  - ∉ - "does not belong to" or "is not an element of"
- The <u>cardinality</u> $|S|$ of a set S is the number of elements in S.
- The <u>empty set</u> ∅ is a set which has no elements.
- The <u>universe</u> U contains everything, and is often regarded as a set.
- Two sets S and T are <u>equal</u> ($S = T$) if
  - i) every element of S is also an element of T, and
  - ii) every element of T is also an element of S.

i.e. when they have precisely the same elements.

Set terminology

- A <u>subset</u> of a set is a part of the set.
  - $\subseteq$ - "is a subset of"
  - $\subset$ - "is a proper subset of"
- S is a subset of T if each element of S is also an element of T.
  - $S = T$ if and only if $S \subseteq T$ and $T \subseteq S$.
- S is a <u>proper</u> subset of T if S is a subset of T and $S \neq T$.
  - $\emptyset$ is a proper subset of any non-empty set.
  - Any non-empty set is an improper subset of itself.
- The <u>power set</u> $\wp(S)$ of a set S is the set containing all the subsets of S.
  - $|\wp(S)|$ = number of subsets of $S = 2^{|S|}$.

Set terminology

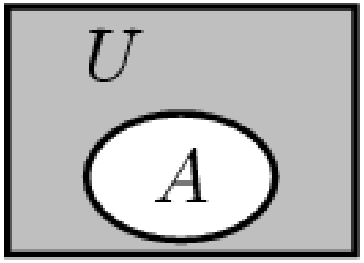
- It is often convenient to assume that elements are linearly ordered by a relation, usually denoted by '<' (read "less than" or "precedes").
- A linear order on a set *S* satisfies two properties:
  - For any *a* and *b* in *S*, exactly one of $a < b$, $a = b$, or $b < a$ is true.
  - For all *a, b*, and *c* in *S*, if $a < b$ and $b < c$, then $a < c$ (transitivity).
- The term multiset or bag is used for a "set with repetitions"

Set operations

- complement (‾) – "not"

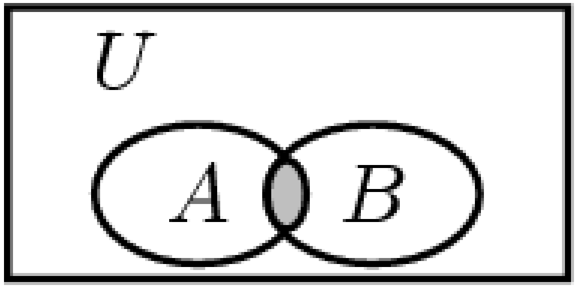
$$\overline{A} = \{x \in U : x \notin A\}$$

$$|\overline{A}| = |U| - |A|$$



- intersection (∩) – "and"

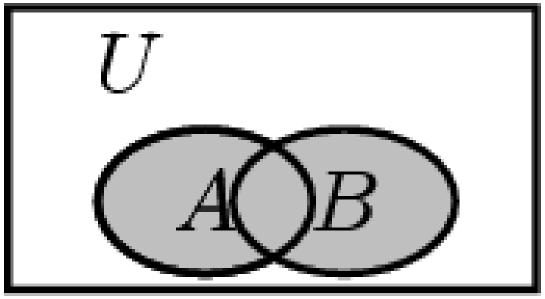
$$A \cap B = \{x \in U : x \in A \wedge x \in B\}$$



- union (∪) – "or"

$$A \cup B = \{x \in U : x \in A \vee x \in B\}$$

$$|A \cup B| = |A| + |B| - |A \cap B|$$

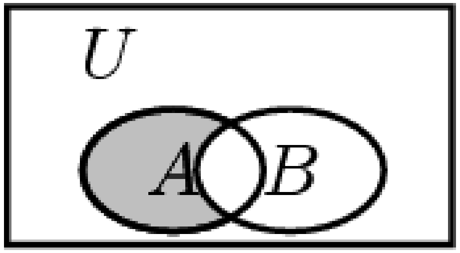


Set operations

- difference (\,−) – "but not"

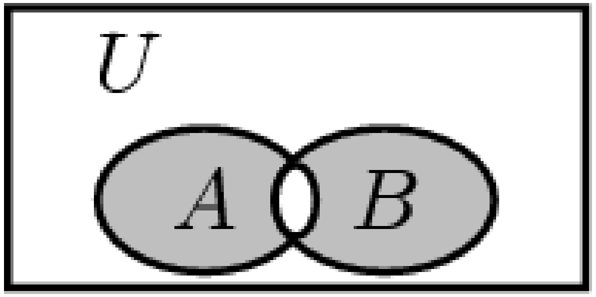
$$A \setminus B = \{x \in U : x \in A \wedge x \notin B\} = A \cap \overline{B}$$

$$|A \setminus B| = |A| - |A \cap B|$$



- symmetric difference (Δ ,⊕) – "exclusive or"

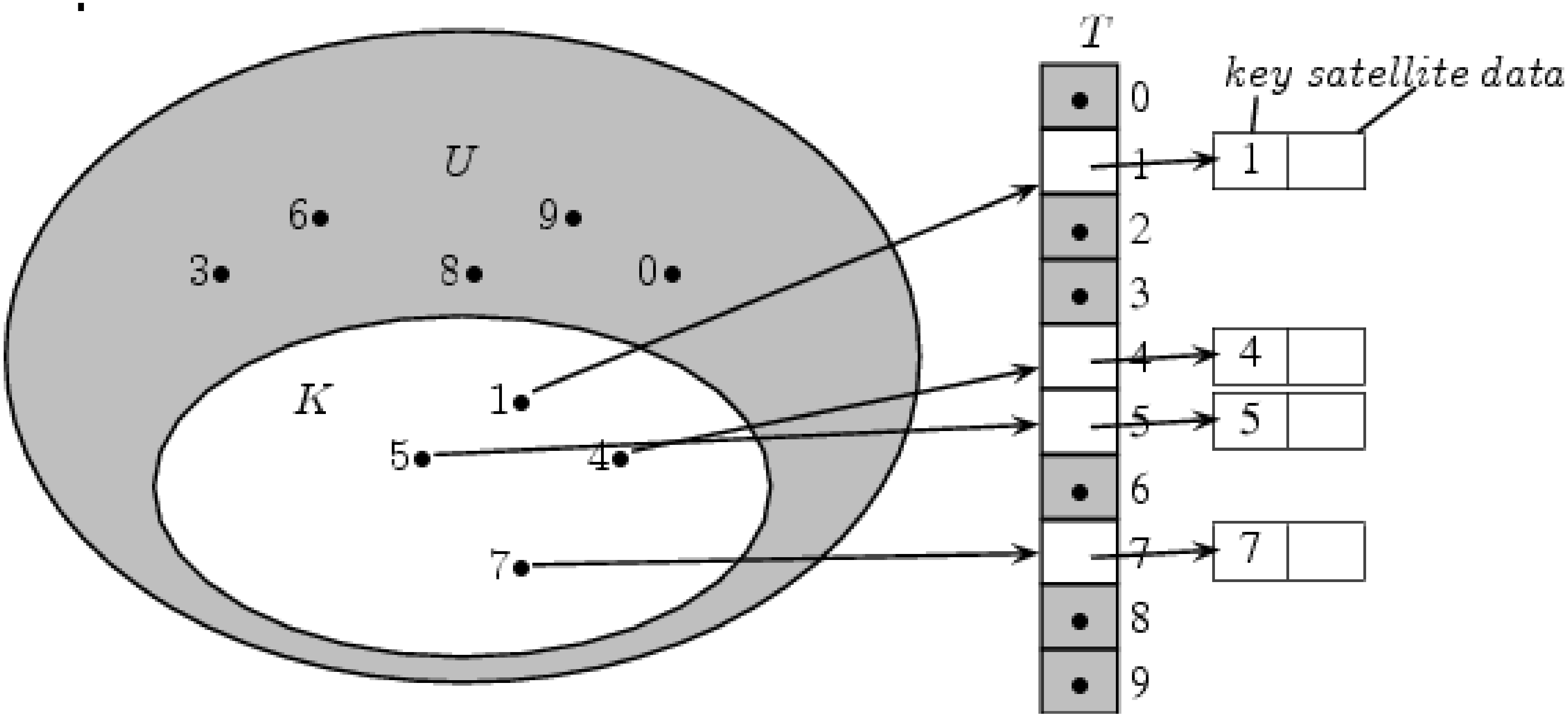
$$A \Delta B = (A \setminus B) \cup (B \setminus A)$$
$$= (A \cup B) - (A \cap B)$$



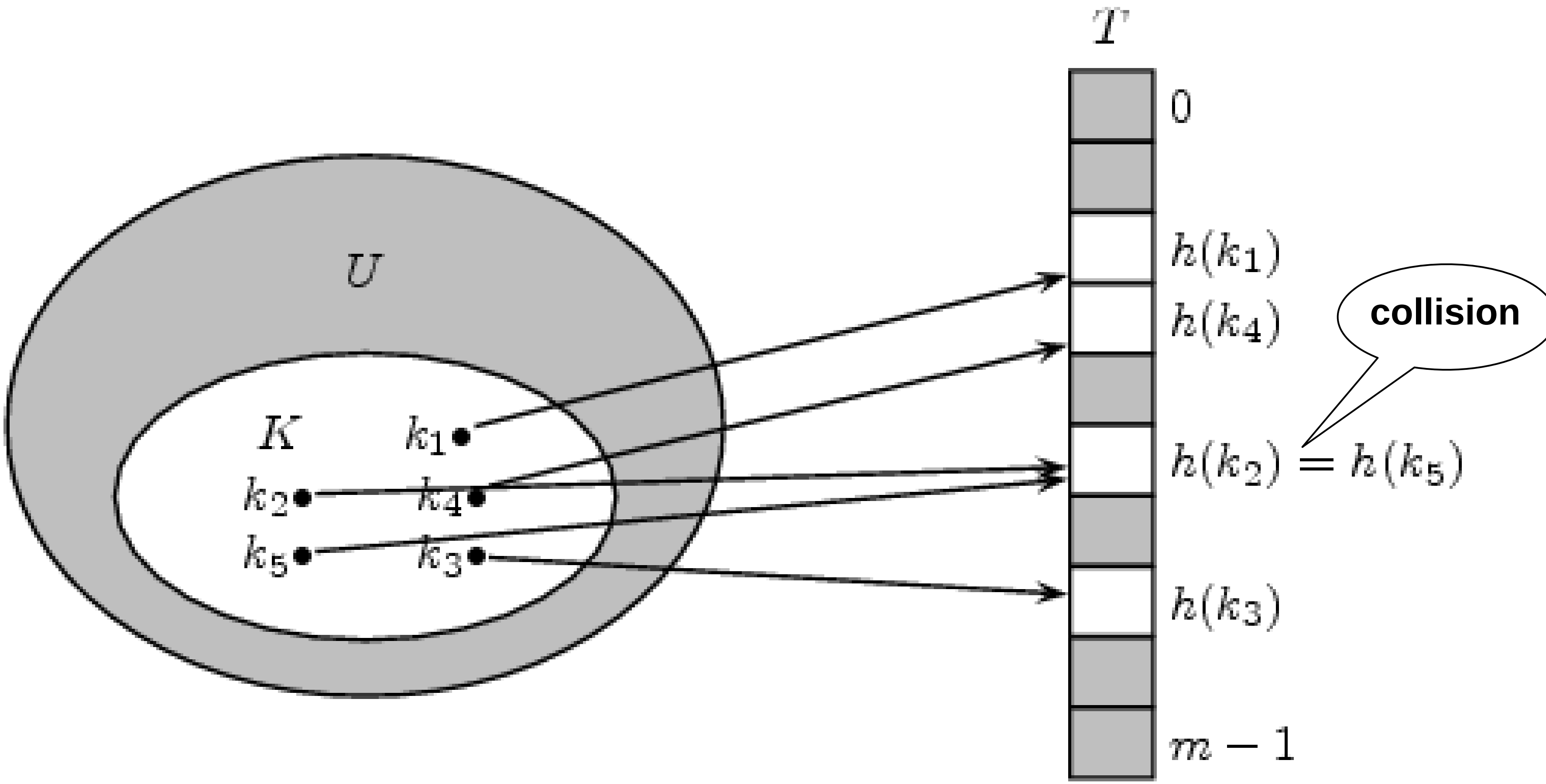
- Two sets A and B are disjoint if A ∩ B = Ø.

Set implementation. Direct access table

- *U*={0, 1, 2, 3, 4, 5, 6, 7, 8, 9} universe of keys
- *K*={1, 4, 5, 7} actual
.
- Direct access table *T*



Set implementation. Hash table



Hash tables. Open addressing

- Open addressing: all elements are stored inside the table (as in the previous example)
  - For insertion we successively examine the hash table looking for an unoccupied slot
  - The slots we check depend on the key we wish to insert

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

Searching in an open addressing hash table

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

Hash table terminology

- A <u>hash function</u> ***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 <u>hash value</u> of key ***x***
- A hash table for a given key type consists of
  - Hash function ***h***
  - Array (called table) of size ***N***

Hash functions

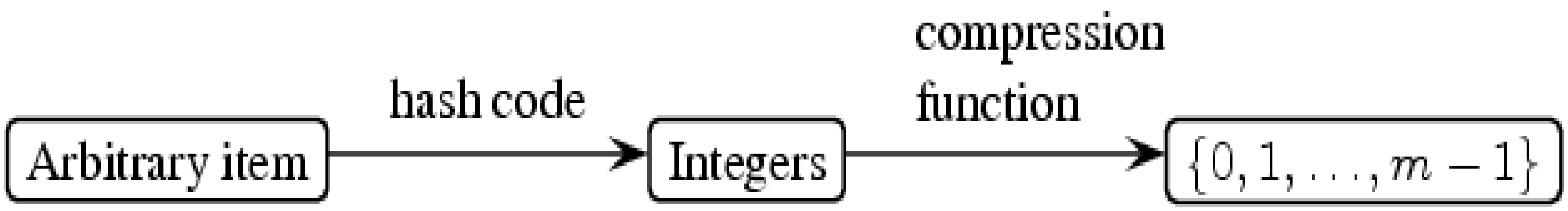
- A hash function is usually specified as the <u>composition</u> of two functions:
  - Hash code:

    h_1: keys → integers
  - Compression function:

    h_2: integers → $[0, N-1]$
- 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 codes

- Memory address:
  - We reinterpret the memory address of the key object as an integer
  - Good in general, except for numeric and string keys
- 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)

Hash codes

- Polynomial accumulation:
  - **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**

$$p(x) = a_0 + a_1\,x + a_2\,x^2 + \ldots$$
$$\ldots + a_{n-1}x^{n-1}$$

  at a fixed value x, ignoring overflows
  - **Especially suitable for strings (e.g., the choice $x = 33$ gives at most 6 collisions on a set of 50,000 English words)**

- Polynomial $p(x)$ 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

$$p_0(x) = a_n - 1$$
$$p_i\,(x) = a_{n-i-1} + x p_{i-1}(z)$$
$$(i = 1, 2, \ldots, n - 1)$$

- We have $p(x) = p_{n-1}(x)$

Compression Functions

- Division:
  - $\boldsymbol{h}_2(\boldsymbol{y}) = \boldsymbol{y} \bmod \boldsymbol{m}$
  - The size $\boldsymbol{m}$ of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $\boldsymbol{h}_2(\boldsymbol{y}) = (\boldsymbol{a}\boldsymbol{y} + \boldsymbol{b}) \mathbf{\ mod\ } \boldsymbol{m}$
  - $\boldsymbol{a}$ and $\boldsymbol{b}$ are nonnegative integers such that $\boldsymbol{a} \mathbf{\ mod\ } \boldsymbol{m} \neq 0$
  - Otherwise, every integer would map to the same value $\boldsymbol{b}$

Rehashing Strategies

- Linear hashing

$$h(k,i) = (h'(k) + i)) \mod m,$$

  - $0 \le \boldsymbol{i} \le \boldsymbol{m}$-1; checks $\boldsymbol{B[h'(k)]}$, then $\boldsymbol{B[h'(k)+1], \ldots, B[m-1]}$
  - **primary clustering** effect: two keys that hash onto different values compete for same locations in successive hashes
- Quadratic hashing

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m,$$

  - $\boldsymbol{h'}$: an auxiliary hash function; $0 \le \boldsymbol{i} \le \boldsymbol{m}$-1;
  - $\boldsymbol{c_1 \ne 0}$ and $c_2 \ne 0$: auxiliary constants
  - checks $\boldsymbol{B[h'(k)]}$**;** next checked locations depend quadratically on $\boldsymbol{i}$
  - **secondary clustering** effect: two different keys that hash onto same locations compete for succcessive hash locations
- Note: ***i*** is the number of trial (0 for first)

Rehashing Strategies

- Double hashing

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m,$$

  - h_1, h_2: auxiliary hash functions; initially, checks position $B[h_1(k)]$ is checked;
  - successive positions are $h_2(k) \bmod m$ away from the previous positions (sequence depends in two ways on key k)
  - $h_2(k)$ and m must be relatively prime (to allow for the whole table to be searched). To ensure this condition:
    - take $m=2^k$ and make $h_2(k)$ generate an odd number or
    - take m prime make $h_2(k)$ return a positive integer m' smaller than m

$$h_1(k) = k \mod m,$$
$$h_2(k) = 1 + (k \mod m'),$$

An Analysis of Open Addressing

- **Assumption: N out of m buckets filled**
- **Probability of initial collision: N/m**
- **Probability of collision after first rehash:** $$\frac{N \times (N-1)}{m \times (m-1)}$$
- **Probability of at least i collisions:** $$\frac{N(N-1)..(N-i+1)}{m(m-1)...(m-i+1)}$$
- **Average number of probes for insertion**

$$1 + \sum_{i=1}^{\infty} \left(\frac{N}{m}\right)^i \approx \frac{m}{m-N}$$

- **The average insertion cost per bucket to fill M of the m buckets**

$$\frac{1}{M} \sum_{N=0}^{M-1} \frac{m+1}{m+1-N} = \frac{1}{M} \int_0^{M-1} \frac{m}{m-x} dx = \frac{m}{M} \ln \frac{m}{m-M+1}$$

- **Conclusion: to fill the table completely (M=m) requires an average of $\ln m$, or $m \ln m$ probes in total**

Hash table performance

- In the worst case, searches, insertions and deletions on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $\alpha = M / m$ 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

Collision Handling by Chaining

- **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

$\textsc{ChainingHashInsert}(B, x)$

insert x at the front of the list $B[h(key[x])]$

$\textsc{ChainingHashSearch}(B, k)$

find element of key k in the list $B[h(key[x])]$

$\textsc{ChainingHashDelete}(B, x)$

delete x from the list $B[h(key[x])]$

Hash Table Animation

- Very good tutorial: http://research.cs.vt.edu/AVresearch/hashing
- http://www.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashApplet.htm
- http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html

Mapping ADT

- **Mapping** (associative store): a function from elements of one type, called the domain type to elements of another type (possibly the same) type, called the range type.
- Operations:
  - createEmpty(A): initializes the mapping A by making each domain element have no assigned range value
  - assign (A, d, r) defines A(d) to be r
  - compute (A, d, r) returns true and sets r to A(d) if A(d) is defined; false is returned otherwise.
- A hash table is an effective way to implement a mapping

Priority Queue ADT

- Priority queue: an ADT based on the set model with the operations:
  - insert and deletemin (as well as the usual createEmpty for initialization of the data structure).
- Additional support operations:
  - min() returns, but does not remove, an entry with smallest key
  - size()
  - isEmpty()
- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key

Priority Queue Entry

- Entry ADT: An **entry** in a priority queue is simply a (key, value) pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Operations for Entry ADT:
  - key(): returns the key for this entry
  - value(): returns the value associated with this entry

Priority Queue Comparator

- Mathematical concept of total order relation ≤
  - Reflexive property: $\boldsymbol{x} \le \boldsymbol{x}$
  - Anti-symmetric property: $\boldsymbol{x} \le \boldsymbol{y} \wedge \boldsymbol{y} \le \boldsymbol{x} \Rightarrow \boldsymbol{x} = \boldsymbol{y}$
  - Transitive property: $\boldsymbol{x} \le \boldsymbol{y} \wedge \boldsymbol{y} \le \boldsymbol{z} \Rightarrow \boldsymbol{x} \le \boldsymbol{z}$
- A <u>comparator</u> encapsulates the action of comparing two objects according to a given total order relation
  - A generic priority queue uses an auxiliary comparator
  - The comparator is external to the keys being compared
  - When the priority queue needs to compare two keys, it uses its comparator
- Comparator main operation:
  - <u>compare</u>(x, y): Returns an integer i such that i < 0 if a < b, i = 0 if a = b, and i > 0 if a > b; an error occurs if a and b cannot be compared.

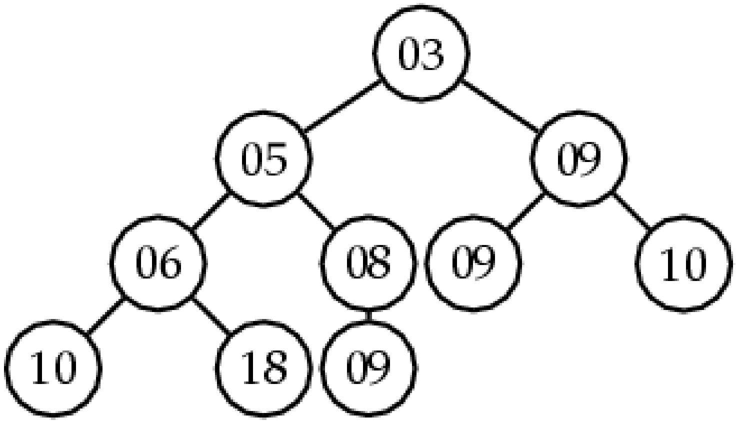
Implementations of Priority Queues

- Unsorted list
- Performance:
  - insert takes $O(1)$ time (we can insert the item at the beginning or end of the list)
  - deleteMin and min take $O(n)$ time (we have to scan the entire list to find the smallest key)

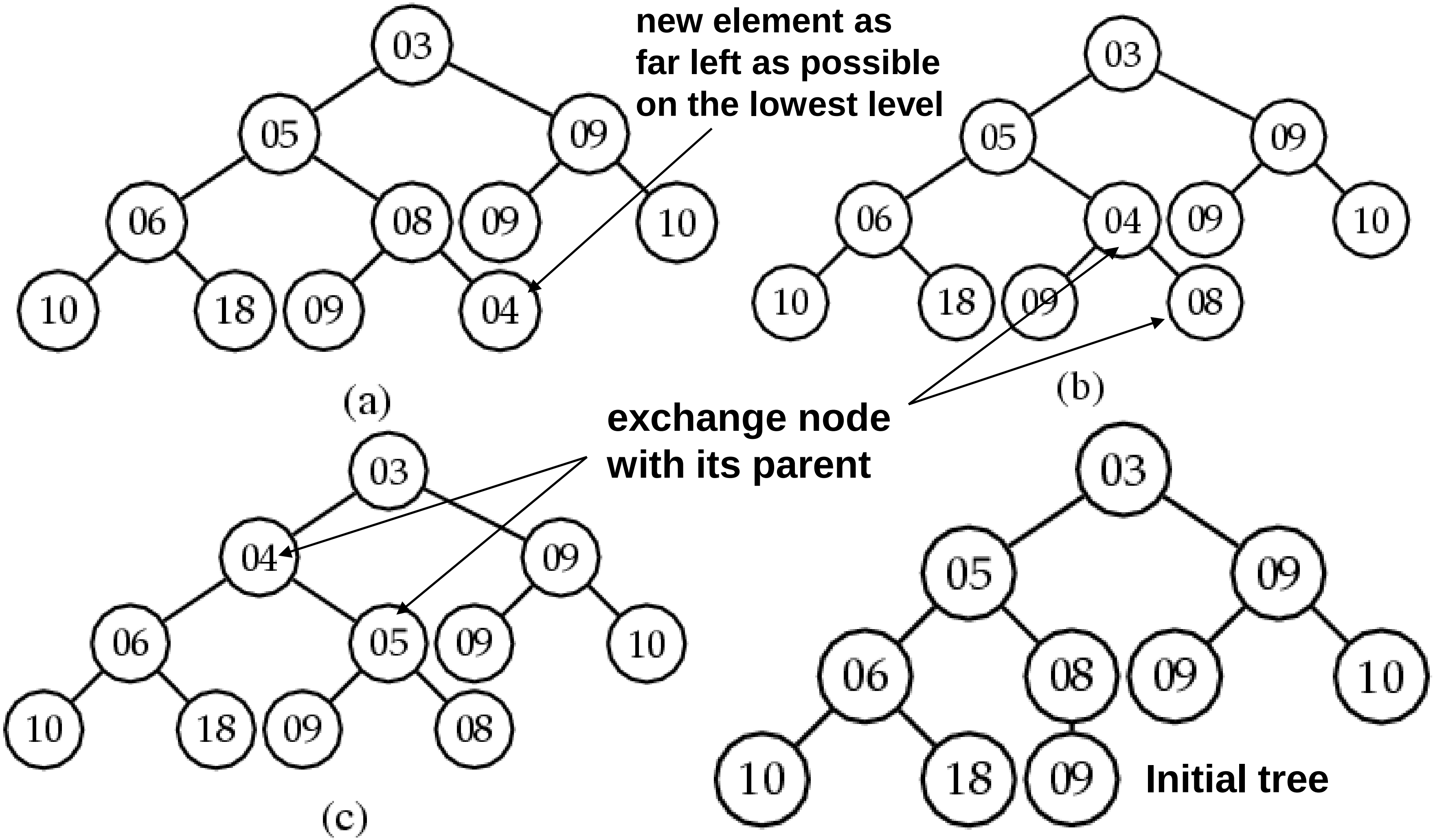
- Sorted list
- Performance:
  - insert takes $O(\mathrm{n})$ time (we have to fiind a place where to insert the item)
  - deleteMin and min take $O(1)$ time (the item is at the beginning of the list)

Partially Ordered Tree (POT) Implementation of Priority Queues

- Partially ordered tree:
  - Binary tree
  - At the lowest level, where some leaves may be missing, we require that all missing leaves are to the right of all leaves that are not on the lowest level.
  - Tree is *partially* ordered: the priority of node *v* is no greater than the priority of the children of *v*



insert in a POT



Complete Binary Trees

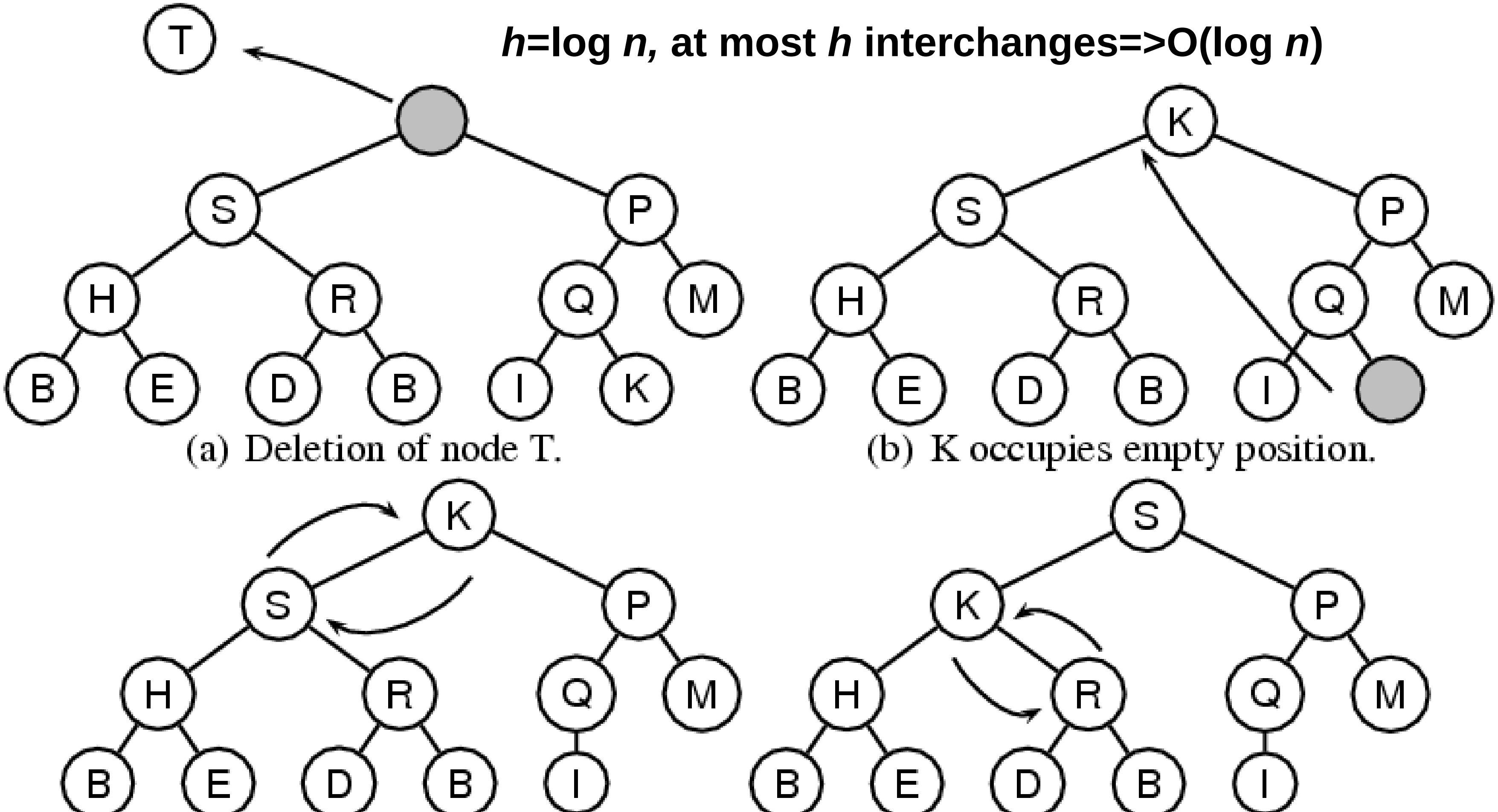
- Complete binary tree of height, *h*, iff:
  - it is empty or
  - its left subtree is complete of height *h*-1 and its right subtree is completely full of height *h*-2 or
  - its left subtree is completely full of height *h*-2 and its right subtree is complete of height *h*-1.
- A complete tree is filled from the left:
  - all the leaves are either on the same level or two adjacent ones, and
  - all nodes at the lowest level are as far to the left as possible.
- Heaps are based on the notion of a complete tree

Heaps

- A binary tree has the heap property if and only if:
  - it is empty or
  - the key in the root is larger than that in either child and both subtrees have the heap property.
- A heap can be used as a priority queue
  - highest (lowest) priority item is at the root: max-heap (min-heap)
  - value of the heap structure: we can both extract the highest (lowest) priority item and insert a new one in O(log n) time

Heap. Deletion of a node (deleteMax)

***h*=log *n*, at most *h* interchanges=>O(log *n*)**



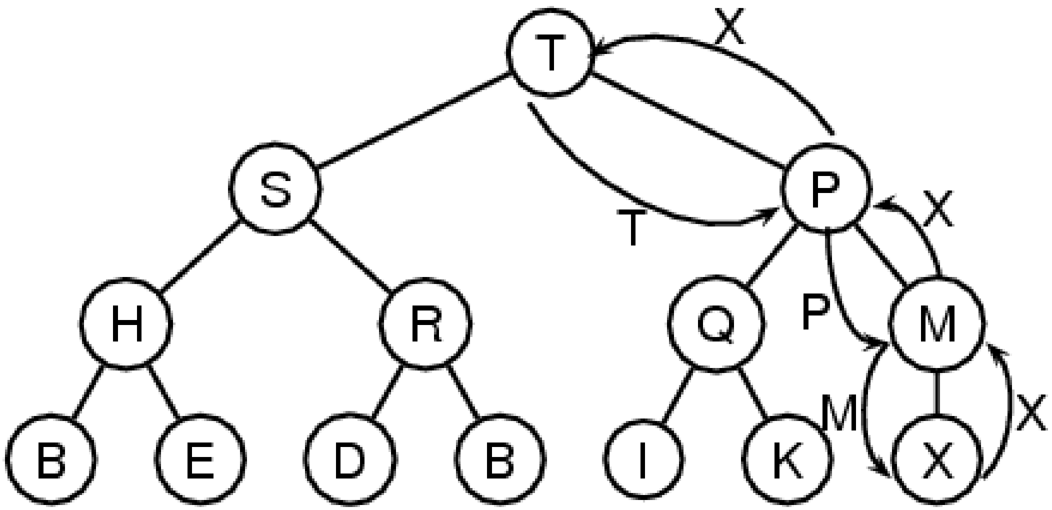
(a) Deletion of node T.

(b) K occupies empty position.

(c) Interchange K with the larger of its children.

(d) Interchange K with the larger of its children.

Heap. Insertion of a node



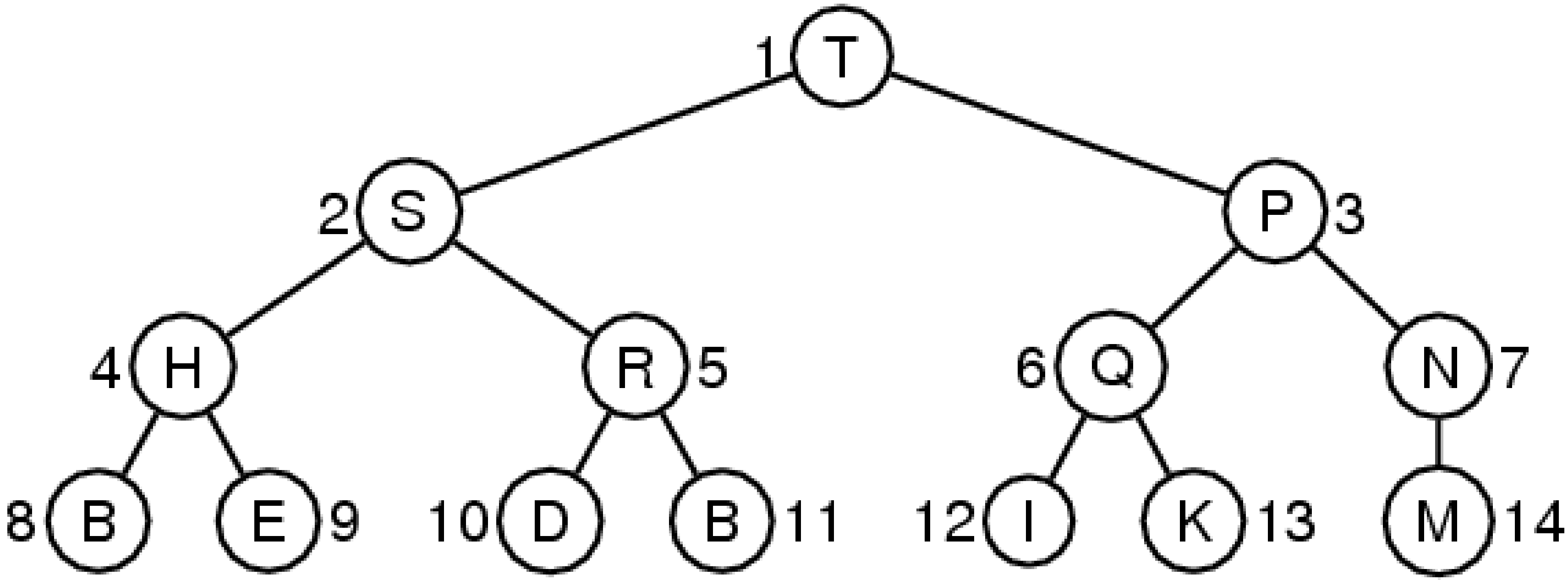
Place new node in the next leaf position and move it up

***h*=log *n*,**
at most *h* interchanges
=>O(log *n*)

- Properties of a complete tree lead to a very efficient storage mechanism using *n* sequential locations in an <u>array</u>

Heap storing in an array

- the left child of node k at position $2k$
- the right child of node k at position $2k+1$



Heap operations

```
HeapExtractMax(A)

if heapSize[A] < 1
    then error "heap underflow"
max ← A[1]
A[1] ← A[heapSize[A]]
heapSize[A] ← heapSize[A] − 1
return max
```

```
HeapInsert(A, key)

heapSize[A] ← heapSize[A] + 1
i ← heapSize[A]
while i > 1 ∧ A[Parent(i)] < key
      do A[i] ← A[Parent(i)]
         i ← Parent(i)
A[i] ← key
```

Heap operations

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heapSize[A] ∧ A[l] > A[i]
    then max ← l
    else max ← i
if r ≤ heapSize[A] ∧ A[r] > A[max]
    then max ← r
if max ≠ i
    then SWAP(A[i], A[max])
         HEAPIFY(A, max)
```

Animation of heap operations

- http://www.cs.auckland.ac.nz/software/AlgAnim/heaps.html
- http://www.cs.auckland.ac.nz/software/AlgAnim/heapsort.html

Summary

- Sets in general
- Abstract data types based on sets
  - operations
  - Implementations
    - lists
- Dictionary ADT
  - Implementations
    - direct access table
    - hash table
- Hash table
- Mapping ADT
  - Implementations
- Priority queues
  - Implementations
- Partially ordered tree
  - Operations: insert, deleteMin
- Heaps
  - Operations: insert, deleteMax
  - Implementation in arrays

Reading

- AHU, chapter 5, sections 5.1, 5.2
- CLR, chapters: 12, 7.1, 7.2, 7.3, 7.5
- CLRS chapter 12, 6.1, 6.2, 6.3, 6.5
- Preiss, chapters: Hashing, Hash Tables and Scatter Tables. Heaps and Priority Queues.
- Notes