Set: Hash, Priority Queue, BST, Trie

Hwansoo Han (SKKU)
Data Structures for Sets

- Set
  - Bit-vector
  - Linked list
  - Hash
  - Priority queue (heap)
  - Binary search tree
  - Balanced tree (2-3 tree, Red-Black tree, B-Tree, ...)
  - Skip list
Abstract Data Type

- UNION (A, B, C) – assign A ∪ B to C
- INTERSECTION (A, B, C) – assign A ∩ B to C
- DIFFERENCE (A, B, C) – assign A - B to C
- MERGE (A, B, C) – disjoint set union, only when A ∩ B = Ø
- MEMBER (x, A) – True if x ∈ A
- MAKENULL (A) – make set A to be a null set
- INSERT (x, A) – make A to be A ∪ { x }
- DELETE (x, A) – make A to be A – { x }
- ASSIGN (A, B) – make A to be equal to B
- MIN (A), MAX (A) – elements in A should be linearly ordered
- EQUAL (A, B) – true if A = B
- FIND (x) – name of the unique set of which x is a member
Bit-Vector Implementation of Sets

- Set by a bit vector
  - $i_{th}$ bit is true, if $i$ is an element of the set
  - A small universal set, whose elements are integers 1, ..., $N$ for some fixed $N$

- Large space, but fast operations
  - MEMBER, INSERT, DELETE in constant time: $O(1)$

- Slow operations for large universal set
  - UNION, INTERSECTION, DIFFERENCE in time proportional to the size of universal set: $O(U)$
    - If the universal set fits in one computer word, a single logical operation performs the set function
Linked-List Implementation of Sets

- Use space proportional to the size of set
  - Items in the list are the elements of the set
  - Bit-vector: proportional to the size of the universal set

- Sorted list to reduce operation cost
  - A: $e_1 < e_2 < e_3 < \ldots < e_n$, B: $e_1 < e_2 < e_3 < \ldots < e_m$
  - MEMBER, INSERT, DELETE: $O(n)$
  - INTERSECTION, UNION, DIFFERENCE can be done without searching the entire list: $O(n+m)$
    - Unsorted list takes $O(nm)$ steps
Hashing
Hashing

- List, array for directory
  - INSERT, DELETE, MEMBER require $O(N)$ for $N$ elements

- Hashing is widely used for directory
  - Operations require constant time on the average
  - In the worst case, could take $O(N)$
Open vs. Closed Hashing

- *Open (or *external*) hashing
  - Use potentially unlimited space
  - No limit on the size of set

- *Closed (or *internal*) hashing
  - Use a fixed space
  - Limit on the size of set
Open Hashing

Main idea
- Partition (possibly infinite) elements into a finite number of classes – e.g. \( B \) classes numbered 0, 1, ..., \( B-1 \)

Hash function, \( h(x) \)
- \( x \) is called *key*
- The value of \( h(x) \) is called *hash value*
- If \( h(x) = i \), \( x \) belongs to \( i_{th} \) class

Maintain \( B \) buckets – 0, 1, ..., \( B-1 \)
- Hash value is one of the *bucket numbers*
- An array, *bucket table* keeps the headers for \( B \) lists
Open Hashing – data organization

- list of elements in each bucket

- bucket table
  - headers

0 → ...
1 → ...
B-1 → ...
Open Hashing – data organization

- If buckets are roughly equal in size
  - The list for each bucket will be short
  - Average bucket will have $N/B$ members

- If we can estimate $N$, choose $B$ to be as large
  - Each bucket will have one or two members
  - Operations will take, on average, small constant time

- Hash function $h$ should distribute fairly evenly among the buckets – i.e. random value independent of $x$
Open Hashing – example $h(x)$

- Hash function for string
  - Add the encoding of each character – $\text{ord}(c)$

```pascal
function h (x: nametype) : 0..B - 1;
var
    i, sum: integer
begin
    sum := 0;
    for i: 1 to 10 do
        sum := sum + \text{ord}(x[i]);
    end
    h := sum \mod B
end { h }
```
Open Hashing - operations

- MEMBER \((x, A)\) – \(x\): element, \(A\): bucket table
  - Search the bucket list headed by \(A[h(x)]\)

- INSERT \((x, A)\)
  - If not MEMBER \((x, A)\), insert \(x\) to the bucket list headed by \(A[h(x)]\)

- DELETE \((x, A)\)
  - Search the bucket list headed by \(A[h(x)]\)
  - If found, delete \(x\) from the list
Closed Hashing

- Keep the elements in the bucket table itself
  - No use of separate lists as in open hashing

- Only one element can be placed in any bucket
  - What if two elements, $x$ and $y$, are needed to be placed in the same bucket — i.e. $h(x) = h(y)$
  - Use *rehash strategy*
    - Find an alternative place for one element, if they have conflicting hash values by using a sequence of rehash functions, $h_1(x)$, $h_2(x)$, ... in order
Closed Hashing - example

- Suppose $B = 8$ and hash values for keys, $a, b, c, d$
  - $h(a) = 3$, $h(b) = 0$, $h(c) = 4$, $h(d) = 3$

- Rehash strategy
  - *Linear hashing*
  - $h_i(x) = (h(x) + i) \mod B$
Closed Hashing - operations

- MEMBER \((x, A)\) – \(x\): element, \(A\): bucket table
  - Examine \(h(x), h_1(x), h_2(x), \ldots\) until an empty bucket found
  - If found before an empty bucket, \(x\) belongs to \(A\)
  - If found empty bucket, \(x\) does not belong to \(A\)

- If deletions are performed, empty bucket does not guarantee \(x\) is not in somewhere else
  - When delete, insert a constant called \textit{deleted} into a bucket
  - Need to distinguish between \textit{deleted} and \textit{empty}
  - \textit{Deleted} can be treated as available space when inserting
Priority Queue
Priority Queue – min (or max)

- List, array implementation of set
  - INSERT, DELETE, MEMBER require $O(N)$ for $N$ elements
  - DELETEMIN (or DELETEMAX) requires $O(1)$
    - $O(N)$, if unsorted list or array

- Hash
  - No support for DELETEMIN (or DELETEMAX)
  - $O(1)$ for INSERT, DELETE, MEMBER

- Need ADT with $O(\log N)$ operations
  - DELETEMIN (or DELETEMAX)
  - INSERT
Priority Queue

- An ADT based on set with operations
  - INSERT
  - DELETEMEMIN
  - MAKENULL

- Priority among elements
  - $p(a)$ for each element $a$, produces a real number, more generally a member of some linearly ordered set
  - DELETEMEMIN returns some element of smallest priority
Priority Queue Implementation

- Can use implementations for set
  - Linked list (sorted or unsorted)
    - Sorted list makes easy to find minimum priority element
    - Unsorted list makes easy to insert
    - One of INSERT and DELETEMIN will take $O(N)$
  - Bit-vector
    - Sequential search of bit vectors to find minimum

- But hash table cannot be used
  - No convenient way to find minimum
Priority Queue Implementation

- Partially ordered trees for priority queues
  - INSERT and DELETEMIN in $O(\log n)$ steps

- Main idea of partially ordered trees
  - Organize elements of the priority queue in a binary tree
  - Make the binary tree as balanced as possible
  - Make the priority of node $v$ in no greater than the priorities of $v$’s children
Partially Ordered Tree - example

- Leaves at the lowest level are filled from the left
Partially Ordered Tree Operations

- **DELETEMIN**
  - Root has the element with minimum priority
  - Thus, remove the root and take the rightmost element at the lowest level and temporarily put it at the root
  - Push down this element as far down as it will go, by exchanging it with the one of its children having the smaller priority – *percolation* step
  - The element goes down either to a leaf or to a node where its priority is no larger than either of its children
Partially Ordered Trees - DELETEMIN

- DELETEMIN (P)

```
  3
 / \
5   9
 / \
6   9
 / \
10  10
 / \
10  18
```

```
  9
 / \
5   9
 / \
6   8
 / \
10  9
 / \
10  18
```
Partially Ordered Trees - DELETEMIN

```
    5
   / \
  9   9
 /   / /
6   8   9 10
```

```
    5
   / \
  9   9
 /   / /
6   8   9 10
```

25
Partially Ordered Trees - INSERT

- INSERT
  - Place the element to insert as far left as possible at the lowest level – if the current level is all filled, start a new level.
  - If the new element has priority lower than its parent, exchange it with its parent
  - Repeat the exchange, until it reaches either at the root or at the position where its priority is larger than its parent
Partially Ordered Trees - INSERT

- INSERT (4, P)
Partially Ordered Trees - INSERT

```
3
   5                      3
  6  4  9  10            9  10
 10 18  9  8             6  5  9  10
```


Partially Ordered Trees - Array Implementation

- **Heap**
  - Array representation for binary trees, balanced as much as possible, and have leaves at the lowest level pushed to the left.
  - For $n$ nodes, use the first $n$ elements of array $A[1..N]$.
  - For a node $A[i]$, left child, if exists, is at $A[2i]$ and right child, if exists, is at $A[2i+1]$.
  - For a node $A[i]$, its parent is at $A[i \div 2]$ ($i > 1$).
Partially Ordered Trees in Heap

- Fill up $A[1], A[2], \ldots, A[n]$ from the top and within a level, from the left.

$$A[1..n] = 3, 5, 9, 6, 8, 9, 10, 10, 18, 9$$
Binary Search Tree
Log$(n)$ Operations for Set

- MEMBER, INSERT, DELETE
  - Bit-vector: $O(1)$, Linked-list: $O(n)$

- General set ADT
  - MEMBER, INSERT, DELETE, DELETEMIN support

- Log$(n)$ – MEMBER, INSERT, DELETE, DELETEMIN
  - Binary search tree
  - Balanced trees (B-tree, Red-black tree, ...)
  - Skip list
Binary Search Tree

- Binary search tree
  - Represents sets whose elements are linearly ordered

- Operations
  - INSERT
  - DELETE
  - MEMBER
  - MIN

- $O(\log n)$ steps on the average for a set of $n$ elements
Binary Search Tree

- Binary tree
  - Two or one child nodes for non-leaf nodes

- Binary search tree
  - Binary trees whose all nodes satisfy the following property
    - All elements stored in the left subtree of node $x$ are all less than the element at node $x$, and
    - All elements stored in the right subtree of node $x$ are all greater than the element at node $x$
Binary Search Tree - Examples
Binary Search Tree

- Each node has
  - Element and two subtrees

```plaintext
type
nodetype = record
  element: elementtype;
  leftchild, rightchild: ↑nodetype
end;
```

- BST set
  - Pointer to the root node of binary search tree

```plaintext
type
SET = ↑nodetype;
```
Binary Search Tree - Operations

- MEMBER

```pascal
function MEMBER ( x: elementtype; A: SET ): boolean,
  { returns true if x is in A, false otherwise }
begin
  if A = nil then
    return (false) { x is never in ∅ }
  else if x = A \element then
    return (true)
  else if x < A \element then
    return (MEMBER(x, A \leftchild ) )
  else { x > A \element }
    return (MEMBER(x, A \rightchild ) )
end; { MEMBER }
```
Binary Search Tree - Operations

- INSERT

```
procedure INSERT ( x: elementtype; var A: SET );
   { add x to set A }
begin
   if A = nil then begin
      new (A);
      A ^.element := x;
      A ^.leftchild := nil;
      A ^.rightchild := nil
   end
   else if x < A ^.element then
      INSERT(x, A ^.leftchild)
   else if x > A ^.element then
      INSERT (x, A ^.rightchild)
   { if x = A ^.element, we do nothing: x is already in the set }
end; { INSERT }
```
Binary Search Tree - Operations

- **DELETEMIN**

```plaintext
function DELETEMIN ( var A: SET ): elementtype;
  { returns and removes the smallest element from set A }
begin
  if A ↑.leftchild = nil then begin
    { A points to the smallest element }
    DELETEMIN := A ↑.element;
    A := A ↑.rightchild
      { replace the node pointed to by A by its right child }
  end
  else { the node pointed to by A has a left child }
    DELETEMIN := DELETEMIN(A ↑.leftchild)
end; { DELETEMIN }
```
procedure DELETE ( x: elementtype; var A: SET );
{ remove x from set A }
begin
if A <> nil then
  if x < A ↑.element
    DELETE(x, A ↑.leftchild)
  else if x > A ↑.element then
    DELETE(x, A ↑.rightchild)
  { if we reach here, x is at the node pointed to by A }
else if (A ↑.leftchild = nil) and (A ↑.rightchild = nil) then
  A := nil  { delete the leaf holding x }
else if A ↑.leftchild = nil then
  A := A ↑.rightchild
else if A ↑.rightchild = nil then
  A := A ↑.leftchild
else  { both children are present }
  A ↑.element := DELETEMIN(A ↑.rightchild)
end;  { DELETE }
Binary Search Tree vs. Hash Table

- **Binary search tree**
  - INSERT, DELETE, MEMBER, DELETEMIN
  - $O(\log n)$ on the average

- **Hash table**
  - Operations except MIN take constant time
  - MIN takes $O(n)$ for $n$ element set
  - If MIN is not needed (or not frequently used), hash table is better
Binary Search Tree vs. PO Tree (or Heap)

- **Binary search tree**
  - **INSERT, DELETE, MEMBER, DELETEMIN**
  - $O(\log n)$ on the average

- **Partially ordered tree (or Heap)**
  - Operations take $O(\log n)$ not only on the average, but also in the worst case – **INSERT, DELETEMIN**
  - But no general **DELETE** is supported
  - **MEMBER** requires $O(n)$
Trie
Trie

- Special structure for sets of character strings
  - Generally, strings of any object types, e.g. integers

- Operations
  - INSERT
  - DELETE
  - MEMBER
  - MAKENULL
  - PRINT
Trie

- **Characteristics of trie**
  - Share the same prefix among multiple words
  - Each path from the root to a leaf corresponds to one word
  - *Endmarker* symbol, $, to the ends of all words
    - To avoid confusion between words THE and THEN
    - No prefix of a word can be a word itself
      - Assume all words are $ terminated
Trie – a trie example

{ THE, THEN, THIN, THIS, TIN, SIN, SING }
Trie – trie nodes

- Characteristics of trie nodes
  - Each node has at most 27 children, A, B, ..., Z, $
  - Most node will have many fewer than 27 children
  - A leaf reached by an edge labeled $ cannot have any children, and may not be there
  - Trie contains no alphabets for a word, but associated nodes

- Trie node
  - Mapping : \{ A, B, ..., Z, $ \} \rightarrow “pointer to trie node”
  - Mapping to the next level trie nodes
  - Mapping of $ can be either nil or a self-pointer
Trie – trie node

Abstract data type TRIENODE

- ASSIGN \((node, c, p)\)
  \(node\): current trie node

- VALUEOF \((node, c)\)
  \(c\): character

- GETNEW \((node, c)\)
  \(p\): pointer to next level trie node

Implementation of TRIENODE

- Array – mapping for 27 characters, A, B, …, Z, $

- List – add mapping when needed
Trie – trie nodes (array implementation)

type
chars = ('A', 'B', ..., 'Z', '$');
TRIENODE = array[chars] of ↑TRIENODE;
TRIE = ↑TRIENODE;

{TRIE is a pointer type to TRIENOE structure }

procedure ASSIGN ( var node: TRIENODE; c: chars; p: ↑TRIENODE );
begin
  node [c] := p
end; { ASSIGN }

function VALUEOF ( var node: TRIENODE; c: chars ) ↑TRIENODE;
begin
  return (node [c])
end; { VALUEOF }

procedure GETNEW ( var node: TRIENODE; c: chars );
begin
  new (node [c]);
  MAKENULL(node [c])
end; { GETNEW }
Trie – operations (array implementation)

**procedure** `INSERT(x: wordtype; var words: TRIE);`

**O(m) algorithm**  

$m = |x|$  

length of word

```
var
  i: integer; { counts positions in word x }
  t: TRIE; { used to point to trie nodes corresponding to prefixes of x }
begin
  i := 1;
  t := words;
  while $x[i] \neq '$' do begin
    if VALUEOF($t\uparrow$, $x[i]$) = nil then
      { if current node has no child for character $x[i]$, create one }
      GETNEW($t\uparrow$, $x[i]$);
      $t :=$ VALUEOF($t\uparrow$, $x[i]$);
      { proceed to the child of $t$ for character $x[i]$, whether or not that child was just created }
      i := i+1 { move along the word $x$ }
    end;
    { now we have reached the first '$' in $x$ }
    ASSIGN($t\uparrow$, '$', t) { self-loop to indicate the end of TRIE (leaf) }
    { make loop for '$' to represent a leaf }
  end; {INSERT}
```
Trie – trie node (list implementation)

- Array implementation of trie nodes
  - Takes a collection of words, $p$ different prefixes
  - $27p$ bytes of storage

- List implementation
  - Small domain, 27 characters
  - Mappings defined for few member of that domain
  - Linked list of the characters for which associated value is not the \texttt{nil} pointer
Summary

- **Data structures for set**
  - **Hash**
    - Member/Insert/Delete: $O(1)$ avg, $O(N)$ worst case
  - **Priority queue**
    - Insert/DeleteMin: $O(\log N)$ both avg and worst case
  - **Binary search tree**
    - Member/Insert/Delete/DeleteMin: $O(\log N)$ avg, $O(N)$ worst case
  - **Trie**
    - Member/Insert/Delete: $O(\text{length of word})$ - efficient for short keys
    - Space efficient structure for large number of keys