Fundamental Algorithms
Lecture #8

Cluj-Napoca

Agenda

- Disjoint Sets
  - Concept & list representation (brief review)
  - Tree representation
- Binomial Heaps & Binomial Trees
  - Def
  - Basic operations
- Fibonacci Heaps
- B trees
  - Def
  - Basic operations

Disjoint Sets

- Collection of dynamic DS $S=\{S_1, \ldots, S_k\}$
- $\exists$n elements (objects) in all k sets ($n \geq k$)
- each set S_i is identified by its *representative* element, $x \in S_i$;
- Basic operations:
  - **MAKE-SET** (x)

    Generates a new set, with a single element => n sets initially, each object has its own set, and it is its own representative el.
  - **UNION** (x, y)

    joins 2 disjoint sets, represented by x and y; builds $S_x \cup S_y$ (and destroys S_x and S_y); the representative becomes any of the 2 representatives;
  - **FIND-SET** (x)

    Returns a pointer to the representative element of the set containing element x.

Disjoint Sets – contd.

n = nb. of objects in the whole S

m = total nb. of operations (MAKE-SET, UNION, FIND-SET)

m>=n (as we have n MAKE-SET operations)

Utility/Applications:

- speeds up execution when we need to find/group items with similar features
- graphs (connected components; MST)
- many other

Disjoint Sets - implementation

- Linked List
- A set = a **linked list**
- representative= the first element (head) of the list
- An object in such a list contains
  - The element from the set;
  - The pointer to the next element in the list (LL)
  - Pointer to the representative (ex: blackboard)
- MAKE-SET(x) – builds a list with a single element O(1)
- FIND-SET(x) – returns the representative O(1)
- UNION(x, y) – adds x's list at the end of y's list;
  - representative = former y's representative
  - all x's elements have to update representative pointer (ex: blackboard)

Disjoint Sets – implementation – contd.

- Worst case: $O(m^2)$ for all operations
  - n MAKE-SET (1 for each element)
  - UNION
    - n times (to get to a single set)
    - $1 + 2 + 3 + \ldots +2 = O(n^2)$ (show on the blackboard)
  - n~m (actually m>n, yet n is linear in m)
  - On average, O(m) for a call of UNION, m calls => $O(m^2)$

Disjoint Sets – implementation increase efficiency

- Update pointers for the shorter lists
- Keep as knowledge their length (similar to Order Statistic Trees)
- **Theorem**: For n objects in LL with weighted union, for m MAKE-SET, FIND-SET and UNION takes O(m + nlgn)
- **Proof**: (check the textbook – identify an **informal** justification)

Forest of Disjoint Sets

- Set = **tree** with root; keep parent pointer
- 1 node = 1 element (=1 obj) from the set
- 1 tree = a set
- The root = *representative* el.
- Basic Implem. ~ to lists (no improvement)
  - MAKE-SET (x) build the tree with root only
  - FIND-SET (x) goes up and return the representative
  - UNION (x, y) Ex: (blackboard)

Forest of Disjoint Sets – Heuristics
(to increase performance)

- UNION by **rank**
  - Similar to weighted unify on lists
  - The tree with less nodes will point to the tree with many nodes
  - Info kept at root level = rank = max height of the tree
  - rank ≅ lg (dim) (is an approximation, not an exact value; a guarantee that value is never exceeded)
- PATH **compression**
  - Within the **Find-Set**, each node on the *search path* will update the parent node to the *representative* (instead of parent), and leave the **rank unchanged**!
  - **Shrink does NOT change rank!** Why? Ex: blackboard

rank ≅ lg (dim) It is an approximation, ONLY

Forest of Disjoint Sets – Heuristics

- Rank[x]

  = max height of the subtree rooted by x

  = nb. of edges on the longest path from x to a leaf

  rank[leaf] = 0
- Find-Set leave ranks unchanged

Forest of Disjoint Sets - Implementation

```
MAKE-SET(x)
p[x] <- x
rank[x]<-0
UNION(x, y)
LINK(FIND-SET(x), FIND-SET(y))
LINK(x, y)
if rank [x] > rank [y]
  then p[y] <-x
  else p[x] <-y
if rank [x] = rank [y]
  then rank [y]= rank [y]+1
```

Forest of Disjoint Sets - Implementation

FIND-SET(x)

```
if x!=p[x]
 then p[x] <- FIND-SET(p[x])
 return p[x]
```

Binomial Trees

- Degree-based augmented trees; denoted B_k
- Degree of the tree = number of descendants of the tree
- Properties of a B_k tree:
  - P1: Degree of the root $(B_k) = k$;
  - P2: Number of nodes $(B_k) = 2^k$;
  - P3: Height$(B_k)=k$;
  - P4: Number of nodes at level i in (B_k) is $C_k^i = \binom{k}{i}$
  - P5: If children of (B_k) are numbered from the left to the right as (k-1, k-2, …, ,0), then child i is a (B_i) tree

Binomial Trees

- Degree-based augmented trees; denoted B_k
- Degree of the tree = number of descendants of the tree
- Properties of a B_k tree:
  - P1: Degree of the root (B_k) = k;
  - P2: Number of nodes (B_k) = 2^k;
  - P3: Height(B_k)=k;
  - P4: Number of nodes at level i in (B_k) is $C_k^i = \binom{k}{i}$
  - P5: If children of (B_k) are numbered from the left to the right as (k-1, k-2, …, ,0), then child i is a (B_i) tree
- Recursive definitions:
  - A B_k tree is 2 B_{k-1} - trees, with their roots linked (picture on the blackboard)
- Be aware of the implication the equivalence definitions following P5 and the recursive definition!

Binomial Trees

- **Recursive definitions:**
  - **A B_k tree is 2 B_{k-1} - trees, with their roots linked (see picture and follow discussion)**
  - **A B_k tree is collection of k trees: B_{k-1}, B_{k-2}, …, B_1, B_0 – trees (see picture and follow discussion)**
- **Proof of P4: Number of nodes at level i of (B_k) is C_k^i**

Binomial Trees

- **Recursive definitions:**
  - **A B_k tree is 2 B_{k-1} - trees, with their roots linked (see picture and follow discussion)**
  - **A B_k tree is collection of k trees: B_{k-1}, B_{k-2}, …, B_1, B_0 – trees (see picture and follow discussion)**
- **Proof of P4: Number of nodes at level i of (B_k) is C_k^i**
  - **Goes by induction**
  - **On level i we have nodes from 2 B_{k-1}trees**
    - **From the first tree (containing the root of B_k) #nodes at level i = C_{k-1}^i**
    - **From the second one #nodes at level i-1 (one level less; level measured from the root) = C_{k-1}^{i-1}**
    - **So there are $C_{k-1}^i + C_{k-1}^{i-1} = C_k^i$ nodes at level i in B_k**

Example

Exemplu 2

Binomial Heaps

- Binomial Heap (H) = A set of Binomial trees with the following properties:
  - P1: each node has a key;
  - P2: each binomial tree in H is heap-ordered (min on top);
  - P3: for any k, there is at most one B_k tree in H.

- Consequence: if H has n nodes, it has at most ? binomial trees.

Binomial Heaps

- **Binomial Heap (H) = A set of Binomial trees with the following properties:**
  - **P1: each node has a key;**
  - **P2: each binomial tree in H is heap-ordered (min on top);**
  - **P3: for any k, there is at most one B_k tree in H.**

- **Consequence: if H has n nodes, it has at most $\lfloor \lg n \rfloor + 1$ binomial trees.**

Binomial Heaps

- **Binomial Heap (H) = A set of Binomial trees with the following properties:**
  - **P1: each node has a key;**
  - **P2: each binomial tree in H is heap-ordered (min on top);**
  - **P3: for any k, there is at most one B_k tree in H.**

- **Consequence: if H has n nodes, it has at most $\lfloor \lg n \rfloor + 1$ binomial trees.**
  - **Justification:**
    - **Max number of trees a H may have = one of each type**
    - **If each type of tree is present, the number of nodes for $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$ is $2^{k-1}, 2^{k-2}, \ldots, 2^0$ respectively**
    - **Nb of nodes of H is their sum = $2^{k-1} + 2^{k-2} + \ldots + 2^0 = 2^k - 1$**
    - **Denote $2^k = n$. H has n nodes, and k trees ($\lfloor \lg n \rfloor + 1$)**

Binomial Heaps – Operations
(|H|=n) (for all, examples on the blackboard)

- **Make-Heap**
  - **Builds an empty Binomial Heap**

Binomial Heaps – Operations
($|H|=n$) (for all, examples on the blackboard)

- **Make-Heap** **O(1)**
  - **Builds an empty Binomial Heap**

- **Binomial-Heap-findMinimum(H)**
  - **Returns the pointer to the root of the B with the min key (NOT removed);**

Binomial Heaps – Operations
(|H|=n) (for all, examples on the blackboard)

- **Make-Heap** **O(1)**
  - **Builds an empty Binomial Heap**
- **Binomial-Heap-findMinimum(H)** **O(lgn)**
  - **Returns the pointer to the root of the B with the min key (NOT removed);**
- **Binomial-Heap-Unite(H1, H2) = merge + links O(lgn)**
  - **merge – merges 2 rooted lists (H1, H2) into a single one sorted by degree (increasing order)**
  - **link – changes a pair of B_{k-1} trees into a B_k tree**
  - **Unite = merge + links from left to right**

Binomial Heaps – Operations

(|H|=n) (for all, examples on the blackboard)

- **Make-Heap** **O(1)**
  - **Builds an empty Binomial Heap**
- **Binomial-Heap-findMinimum(H)** **O(lgn)**
  - **Returns the pointer to the root of the B with the min key (NOT removed);**
- **Binomial-Heap-Unite(H1, H2) =**
  merge + links O(lgn)
  - **merge – merges 2 rooted lists (H1, H2) into a single one sorted by degree (increasing order)** **O(lgn)**
  - **link – changes a pair of B_{k-1} trees into a B_k tree** **O(1)**
  - **Unite = merge + links from left to right** **O(lgn)+lgnO(1)**

Binomial Heaps – Operations

(|H|=n) (for all, examples on the blackboard)

- **Binomial-Heap-extractMinimum(H)**
  - **Binomial-Heap-findMinimum(H)**
  - **removes that tree from H**
  - **Make a heap out of the binomial tree containing the min key =>H1in**
  - **Binomial-Heap-Unite(H, H1)**

Binomial Heaps – Operations

(|H|=n) (for all, examples on the blackboard)

- **Binomial-Heap-extractMinimum(H) O(lgn)**
  - **Binomial-Heap-findMinimum(H) O(lgn)**
  - **removes that tree from H O(1)**
  - **Make a heap out of the binomial tree containing the min key =>H1in O(lgn)**
  - **Binomial-Heap-Unite(H, H1) O(lgn)**

- **Binomial-Heap-keyDelete(H1, H2)**
  - **As if we were to extract min.**
  - **How?**
    - **Decrease the key to delete to -∞+**
    - **restore the heap property of the Binomial tree +**
    - **Extractmin**

Binomial Heaps – Operations

($|H|=n$) (for all, examples on the blackboard)

- **Binomial-Heap-extractMinimum(H) O(lgn)**
  - **Binomial-Heap-findMinimum(H) O(lgn)**
  - **removes that tree from H O(1)**
  - **Make a heap out of the binomial tree containing the min key =>H1in O(lgn)**
  - **Binomial-Heap-Unite(H, H1) O(lgn)**

- **Binomial-Heap-keyDelete(H1, H2) O(lgn)**
  - **As if we were to extract min.**
  - **How?**
    - **Decrease the key to delete to -∞+ O(1)**
    - **restore the heap property of the Binomial tree + O(lgn)**
    - **Extractmin O(lgn)**

Fibonacci Heaps

- Collection of ordered trees
- Properties: like Binomial Heaps with some constraints added/removed.
- Relaxed constraints:
  - May contain several trees of the same degree
  - Rooted, yet unordered
- Added constraints:
  - Children at a given level in a tree are linked to each other (left/right) in a circular, doubly linked list (child list)
  - Node augmentation: degree[x] = number of children in the child list of x
  - The heap maintain a pointer (min[H]) to the root of the tree containing the min key.
- Operations:
  - Like for Binomial Heaps (Hw)
  - Obs: due to relaxations, operations faster:
    - Insert node – a node which is a tree, at the top level
    - Find min – has a pointer
    - Union – simpler, since they are not ordered

Binomial/Fibonacci Heaps Comparative analysis

Operation	Binomial	Fibonacci
Make heap	O(1)	O(1)
Insert	O(lgn)	O(1)
Find min	O(lgn)	O(1)
Extract min	O(lgn)	O(lgn)
Union	O(lgn)	O(1)
Decrease key	O(lgn)	O(1)
delete	O(lgn)	O(lgn)

B-trees

- Previous DS reside in the primary memory
- **Trees on secondary storage devices (disk)**
- A node may have many children
- Goal: **decrease the number of pages accessed** when search for a node
- Store a very large number of keys
- Maintain the height of the tree under control (h very small)

B-trees – contd.

- **Typical pattern while working with B-trees:**

```
x<-pointer to some object
Disk-Read(x)
Operations that access/modify some fields of x
Disk-Write(x)
```

- **Once in memory, operations are performed fast**
- **Objective: as few pages read/write operations**

B-trees – contd.

Generalization of BST (with ordered lists)

- P1: n[x] keys in node x
- P2: keys are ordered

$key_1[x] <= key_2[x] <= \ldots <= key_{n[x]}[x]$

- P3: An internal node contains n[x]+1 pointers to the children $c_i[x]$
- P4: is a search tree:

$key[c_1[x]] <= key_1[x] <= keyc[_2[x]] <=$

- P5: All leaves are at the same level = height of the tree = h
- P6: t = degree of the tree; min(t)=2. Every node (except for the root) has at **least *t-1 keys***, and ***t children***
- P7: Every node (except for the root) has at **most *2t-1 keys***, and ***2t children***

B-trees – Search

- **One pass procedures (top-down ONLY, NO back up; as opposed to PBT, AVL, RB trees)**
- **For ALL operations, the process is JUST top->down!!!**
- **Search**
  - Straightforward generalization of BST search
  - Combined with ordered list search
  - #pages accessed (worst) O(h)
  - Access time in a page (worst) O(t)
  - **Overall O(th)**

B-trees – Insert

- **One pass procedures (top-down ONLY,** NO back up updates; as opposed to PBT, AVL, RB trees**)**
- **Insert**
  - **Search** for a LEAF position to insert
  - Insertion is performed in an **EXISTING** leaf
  - Along the path while searching (top-down), ensure **there is space for a safe insert** (*split full nodes* on the path down to avoid overflowing, so that the insertion is successful in an existing node!!!)
  - A key added to a **full node** (leaf included) will induce:
    - the ***migration of the median key to the parent node,***
    - and the **split** of the given node into 2 nodes (leaf into 2 leaves)
  - **Time: O(th)**
    - **O(h) disk accesses,**
    - **O(t) CPU time in one page**

B Trees - insert

- Like in any BST tree insert in a leaf (in a leaf, NOT as leaf; the node is NOT now created!)
- Stages:
  - **Search** the path for the position (leaf) to insert
  - Ensure the search **path is safe**
  - **Insert** the key in the corresponding **leaf**
- Types of nodes to store a key:
  - Leaf (key to be inserted)
  - Non-leaf (the "safe path" step = in the attempt to make room = in the split stage with median migration up)
- Possible issues
  - Attempt to store in a full node, with (2t-1) keys – issue – node overflows! Not allowed.
- Cases to analyze
  - Not overflowing node – no issue
  - Overflowing node – issue – needs a strategy to handle it

B Trees -insert

- **Strategy**
  - Along the searched path, ANY full node along the path (with already (2t-1) keys) is “fixed” (allowing for a potential full node to accept a new key to be added):
    - **Divide the full node** in 2 nodes with (t-1) keys
    - The **median key** in the full node is **promoted to the parent** node (there *is room*, as we proceed top-down, and an upper node was “fixed”, is not overflowing)
    - if ***root is full*, increase the height** (by adding 1 more node = new root); the ONLY case of *height increase*.
- Insert procedure
  - Top-down approach (descendent)
  - There is NO operation performed on return (bottom up)

B-trees – Delete

- **One pass procedures** (**top-down**, NO back up update; as opposed to AVL, RB trees)
- **Delete**
  - **Search** for the node to contain the key to be removed and identify its type (leaf/no leaf – all nodes are either internal, or leaves; the tree is complete) (z from BST)
  - Physical ***removal*** of one object which *belongs to a leaf* (y in BST)
    - Q: Why y, a node with one successor only in a BST is in a leaf in a B-tree?
  - Along the path while searching (top-down), **ensure the constraints for a safe delete are met** (*merge nodes with degree t* on the path down to avoid underflowing, so that the deletion is possible)
  - **Time: O(th)**
    - **O(h) disk accesses**
    - **O(t) CPU time in one page =>**

B Trees – delete

- **Like in ANY other BST tree**
  - **Search for the key to remove** (pointed by **z**)
  - **If in leaf, delete it**
  - **If not in the leaf, remove (physically) a node** (pointed by **y**; its content is moved in the node pointed by z) **with one-single child** (pointed by **x**) **- the predecessor/successor – (in B-trees y is in a leaf, hence x points to nil)**
- **Types of nodes containing the key to be deleted**
  - **Leaf**
  - **Non-leaf (i.e. internal)**

Node type	Capacity
Leaf	Does not underflow
Non-leaf	Underflows

- **Possible issues**
  - **Attempt to delete from a node with only (t-1) keys – issue – node underflows! Not allowed**
- **Cases to analyze**

B Trees – delete

- **Cases to analyze**

Issue: attempt to delete from a node with only (t-1) keys – node underflows! Not allowed

Node type	Capacity
Leaf	Does not underflow
Non-leaf	Underflows

- **Solution**
  - Similar strategy as in case of insert: **prevent, rather than repair**
  - In the **search stage (for the key to delete**), ensures that **each** node (along the searched path) has the ability to allow for delete (does not underflow)
  - **Along the searched path, any node with only (t-1) keys is "fixed"**
    - If any of the **sibling** nodes has **at least t keys, "borrow"** a key from it (promote the last/first key from the left/right brother node to the parent node, and move down the appropriate key from parent to the almost underflowing node). (see examples for case 3.1 – next slide)
    - If **both sibling** neighbors have **only (t-1) keys, merge** the underflowing node with one of the 2 siblings, and **put the key from the parent node in between them in the new generated (by merge) node.** (see examples for case 3.2.1 next slide). Maybe a height shrink occurs (see examples for case 3.2.2 next slide).
- **Delete procedure**
  - Top-down approach (descendent)
  - There is no operation performed on return (bottom up)

B Trees –delete contd.

Node type	Capacity
Leaf	Does not underflow
Non-leaf	Underflows

- Cases – only 3 to be considered (not 4): non-leaf/underflow is not considered (since along the path, the underflow situation is solved, anyway).
- Cases: (follow the examples – blackboard)
  - Case1 – key in leaf, node does not underflow
    - Simply delete the key
  - Case2 – key not in leaf, node does not underflow
    - Remove pred/succ (from a leaf for a BT) like in BST. The case reduces to case 1 or 3.
  - Case3 – key in leaf, node underflows
    - 3.1 sibling consistent (at least one sibling neighbor has at least t keys) sol: "borrow" from sibling
      - Promote the last/first key in consistent neighbor to parent
      - Move down the key in parent node in between the leaf and consistent sibling to the underflowing leaf
    - 3.2 neighbor siblings both with just (t-1) keys
      - Merge the underflowing leaf with one sibling neighbor by adding in the middle the key in the parent in between the leaf and sibling
        - 3.2.1 keep the height of the tree
        - 3.2.2 *decrease the height* of the tree (if the *parent* is the *root* and has just one single key)

Required Bibliography

- From the Bible – Chapter 21 (Data Structures for Disjoint Sets), Chapter 18 (B-Trees), Chapter 19 (Fibonacci Heaps)
- From the Bible, second edition – Chapter 19 (Binomial Heaps)