New Bulgarian University

Department: Informatics
Field of study: Networking technologies
Course: NETB223 Data Structures Project

Lecturer: Associate Prof. Ph.D. Nikolay Kirov

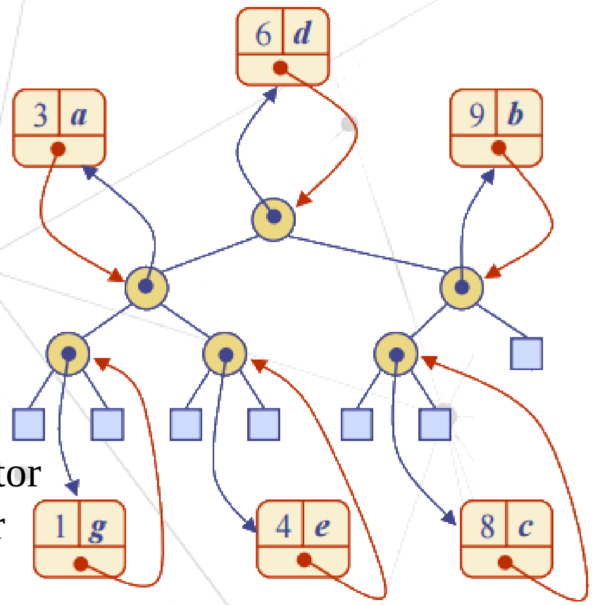
Variant 05:

Locator-Based Search Trees
External Searching
(a,b) Trees,
Update Operations
B-Trees

Student: Georgi Klincharov F45686

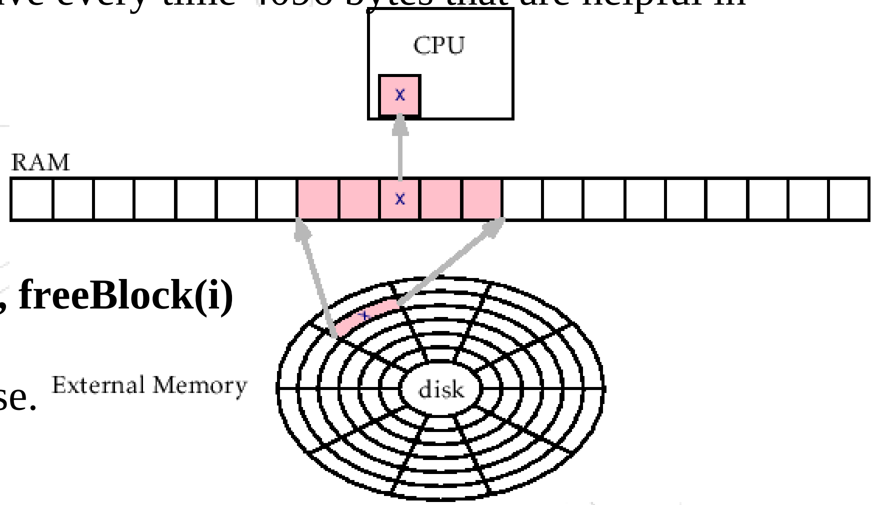
Locator-Based Search Trees

- A locator identifies and tracks an item, consisting of a (key, element) pair
- **Locators** > Positions, because positions change, and a locator sticks with a specific element, even if it changes position in the data structure
- **Locators** are intuitive – reservations, claim queue, airport cancellation queue…
- **Locator-based** Search Tree methods differ from **Dictionary** methods only in that search and insertion operations return an object of type **Locator** and additional operations **remove()** and **replaceKey()** are supported.
- **Locator** ADT methods include:
  - key() – returns the key of the associated item
  - element() – returns the element of the associated item
  - isNull() – determine if it's a null locator
- **Locator-based** Dictionary methods:
  - first(), last() – return a locator to the smallest/largest item
  - locFind(**k**), locFindAll(**k**) – return locator, iterator of locators of the items = **k**. Null locator can be returned
  - locInsertItem(**k, e**) – insert an item (**key, elem**) and return its locator
  - before(p), after(p) – return a locator to an item whose key is larger or smaller than **p.key()**. Null locator can be returned.
  - swapInternal(**p, q**) – instead of copying we relink the nodes



External Searching

- It's a false assumption, that our computer has enough RAM to store everything needed.
- Data must be stored on external mediums: HDD, SSD, File server
- Accessing external storage is extremely slow:
  - HDD access time ~ 19ms * 160 000 = RAM speed
  - SSD access time ~ 0.3ms * 2500 = RAM speed
  - RAM access time ~ 0.000113ms
- HDDs/SSDs have blocks of 4096 bytes. That's a lot of transfer if we need just one byte.
- Transferring each block takes constant time, and computations in internal memory are technically considered free.
- If we organize our data structure carefully, we can have every time 4096 bytes that are helpful in completing whatever operation we are doing.



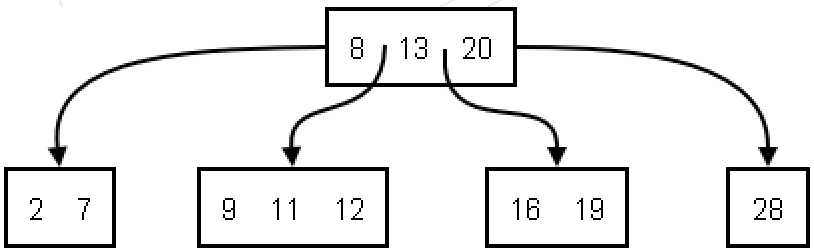
- **We need a more flexible data structure!**

- Supported operations in a **BlockStore** are:
  - **readBlock(i), writeBlock(i, b), placeBlock(b), freeBlock(i)**
- In addition, a **BlockStore** could keep a list of blocks that are freed by **freeBlock(i)** and are available for use.

(a, b) Trees

- **Search Trees** are a **generalization** of **(a, b) Trees**.
- **(a, b) Trees** are **balanced** (all leaves are on the same level) and hold between **a** and **b** children, except the **root**. These are the **Depth** and **Size Properties.**
- Inheriting a general structure we have more **flexibility**.
- The running times relies on the parameters **a** and **b**. We select them so that we minimize disk access.

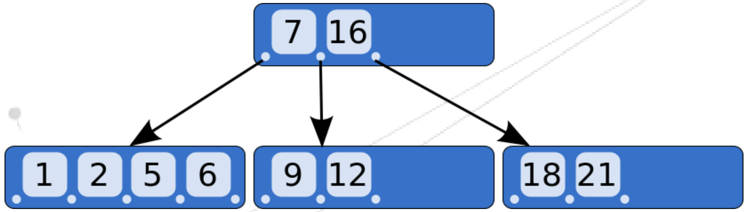
- **Update Operations**:
  - **Inserting** an item could cause an **overflow** and the **b-node** could become **illegal (b+1)**.
    - To **balance** the tree, we split the node and move its middle item into the parent node.
    - **Overflow** in the parent nodes might occur.
  - **Removing** could also cause an **underflow**, rendering the node **illegal (a-1)**.
    - In that case we either perform a transfer with a sibling, that is not an **a-node** or we perform a fusion with an **a-node** resulting a **(2a – 1) node**.



An example of an (2, 4) Tree.

B-Trees

- **(a, b) Trees** are a generalization of the **B-Trees** data structure, which is best known for maintaining a dictionary in external memory.
- A **B-tree of order d** is namely an **(a, b)** tree with **a = (d/2)** and **b = d**.
- The update methods are the same as the already discussed of **(a, b)** trees.
- We can chose **d** so that the **d** children and the **d-1** nodes can all fit into one single disk block. Search and update operations need only one disk transfer to access a node = **O(1)** time.
- If an **overflow** occurs, having **d+1** children, it is split into two nodes that each have **[(d + 1) / 2]** children. The process is repeated at the next level up.
- If an **underflow** ([d/2] -1 children) occurs, we move children from a sibling node having at least [d/2] + 1 children. Or we need to perform a **fusion** with the node's sibling and perform this operation at the parent.
- The requirement that each internal node has at least **d/2** children implies that each disk block is at least half full.



A B-tree of order 2.