Supplement C

Balanced Trees
12.1

Balanced Search Trees
2–3 Tree: Overview

- Key < 50
- 50 < Key < 90
- Key > 90

All leaves at the same level (full basically)
2–3 TreeNode : Implementation

class TreeNode {
private:
    TreeItemType S, L;
    TreeNode* left, mid, right;
    friend class Tree23;
};

left  S  mid  L  right

left  S  mid ?  *

one–value node

two–value node
2-3 Tree: inorder traversal

10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160
2-3 Tree: traverse algorithm

```c
inorder (tree23:Tree23) {
    // Perform a traversal in sorted-key order
    if (tree23's root node r is a leaf)
        visit the data item(s)
    else if (r has two data items) {
        inorder(left subtree)
        visit small data item
        inorder(mid subtree)
        visit large data item
        inorder(right subtree)
    }
    else {
        inorder(left subtree)
        visit small data item
        inorder(right subtree)
    }
}
```

The use of small and large is because we are using a 2-3 tree that has its value sorted by key value.
Retrieve : 140
2–3 Tree: retrieve algorithm

```c
retrieve(tree23:Tree23, key:KeyType, out item:TreeItemType):bool {
  if (key is in root node r) {
    item = data portion of r
    return true
  } else if (r is a leaf) {
    return false;
  } else if (r has two data items) {
    if (key < smaller key of r) {
      return retrieve(left subtree, key, item)
    } else if (key < larger key of r) {
      return retrieve(mid subtree, key, item)
    } else {
      return retrieve(right subtree, key, item)
    }
  } else {
    if (key < r's key) {
      return retrieve(left subtree, key, item)
    } else {
      return retrieve(right subtree, key, item)
    }
  }
}
```
A bst looses balance

After inserting 39, 38, ... 32
A 2–3 tree maintains balance

After inserting 39, 38, ... 32
2–3 : inserts in a leaf node
2–3 : There are three possible splits when inserting

- Split a leaf node
- Split an internal node
- Split the root node
Split a leaf node
Split an internal node
Split an internal node

\begin{center}
\begin{tikzpicture}[level distance=1.5cm,sibling distance=1.5cm, every node/.style={draw,circle,fill=blue!20,minimum size=1cm}]
    \node (P) at (0,0) {P}
    child {node {S \nodepart{m} M \nodepart{l} L}
        child {node {a}}
        child {node {b}}
        child {node {c}}
        child {node {d}}
    }
    child {node {e}};
\end{tikzpicture}
\hspace{1cm}
\begin{tikzpicture}[level distance=1.5cm,sibling distance=1.5cm, every node/.style={draw,circle,fill=blue!20,minimum size=1cm}]
    \node (MP) at (0,0) {MP}
    child {node {S}}
    child {node {L}};
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}[level distance=1.5cm,sibling distance=1.5cm, every node/.style={draw,circle,fill=blue!20,minimum size=1cm}]
    \node (P) at (0,0) {P}
    child {node {S \nodepart{m} M \nodepart{l} L}
        child {node {a}}
        child {node {b}}
        child {node {c}}
        child {node {d}}
    }
    child {node {e}};
\end{tikzpicture}
\hspace{1cm}
\begin{tikzpicture}[level distance=1.5cm,sibling distance=1.5cm, every node/.style={draw,circle,fill=blue!20,minimum size=1cm}]
    \node (PM) at (0,0) {PM}
    child {node {S}}
    child {node {L}};
\end{tikzpicture}
\end{center}
Split the root node
Inserting 39

Before insertion:

50

30

10 20 40

70 90

60 80 100

After insertion:

50

30

10 20 39 40

70 90

60 80 100
Inserting 38
Inserting 37
Inserting 36
Inserting 35
Inserting 34
Inserting 33
Inserting 32...
Inserting 32
Inserting 32...
2–3 : deletes in a leaf node
2–3 : There are five possible outcomes when deleting

- Redistribute a leaf node
- Merge a leaf node
- Redistribute an internal node
- Merge an internal node
- Delete the root
Redistribute a leaf node
Merge a leaf node
Redistribute an internal node
Merge an internal node
Delete the root
Deleting 70
Deleting 80

Before deletion:
- 50
- 30
- 80
- 10, 20, 40, 60, 90

After deleting 80:
- 50
- 30
- 90
- 10, 20, 40, 60, 80

After deleting 50:
- 30
- 50
- 30, 50
- 10, 20, 40, 60, 90
2-3-4 Tree: Overview

- 2-node
  - \( r \)
  - \( T_L \)
  - \( T_R \)

- 3-node
  - \( S \)
  - \( L \)
  - \( T_L \)
  - \( T_M \)
  - \( T_R \)

- 4-node
  - \( S \)
  - \( M \)
  - \( L \)
  - \( T_L \)
  - \( T_M \)
  - \( T_MR \)
  - \( T_R \)
class TreeNode {
private:
    TreeItemType S, M, L;
    TreeNode* left, lMid, rMid, right;
    friend class Tree234;
};
2–3–4 : You insert in a leaf node, but you split 4-nodes en route

This assures there is room in the parent to add a key
2–3–4: There are three possible scenarios when splitting a 4–node

- Be the root
- Have a 2–node parent
- Have a 3–node parent
Be the root
Have a 2-node parent
Have a 3-node parent
Inserting 20
Inserting 50

![Tree Diagram]

- Left Tree:
  - 30
  - 10, 20
  - 60

- Right Tree:
  - 30
  - 10, 20
  - 50, 60
Inserting 40
Inserting 70
Inserting 80
Inserting 15
Inserting 90
Inserting 100
2–3–4: You delete in a leaf node, but you transform 2–nodes into 3 or 4–nodes en route.

This assures that a leaf can borrow a key from its parent.
2–3–4 : There are three possible transformations when deleting

- 1-key sibling and 1-key parent
- 2-key sibling and 2-key parent
- 2-key sibling and 3-key parent
1-key sibling and 1-key parent
1–key sibling and 2–key parent
1-key sibling and 3-key parent
Delete 50

Immediate successor

57
Delete 60
Delete 40
Red–Black Tree: Overview

Represent a 2–3–4 tree as a binary tree

saves storage over 2–3–4

2 reds for 4-node

1 red for 3-node

OR
2–3–4 conversion to Red–Black
Pre-order traversal, with M or L split outs
2–3–4 conversion to Red–Black

Pre-order traversal, with M or L split outs
2–3–4 conversion to Red–Black

Pre–order traversal, with M or L split outs
2–3–4 conversion to Red–Black

Pre-order traversal, with M or L split outs
2–3–4 conversion to Red–Black

Pre-order traversal, with M or L split outs
enum Color {RED, BLACK};

class TreeNode {
private:
    TreeItemType item;
    TreeNode* left, right;
    Color leftColor, rightColor;
    friend class TreeRedBlack;
};
R-B : Insert/Delete requires just a color change
Insertion: split the root

2-3-4 split

r-b conversion

color change
Left: split w/2-node parent
Right: split w/2-node parent

2-3-4 split

2-3-4 to r-b

Color change
Left: split w/3–node parent

2-3-4 split

2-3-4 to r-b

color
Left: split w/3-node parent

2-3-4 split

2-3-4 to r-b

rotation-color
Middle: split w/3-node parent

- 2-3-4 split
- 2-3-4 to r-b
- Rotation-color
Middle: split w/ 3-node parent

```
       P Q
      /   \
     S   M   L
    / \   / \
   b   c  d   e

       P Q
      /   \
     S   M   L
    / \   / \
   b   c  d   e

       P Q
      /   \
    S   M   L
    /   / \
  a   b   c   d

       P Q
      /   \
     S   M   L
    /     \
   a     b
```

2-3-4 split

2-3-4 to r-b

rotation-color

```
       M
      /   \
     P   Q
    /     \
   a     b

       M
      /   \
     P   Q
    /     \
   a     b
d to r-b

       M
      /   \
     P   Q
    /     \
   a     b
```
Right: split w/3-node parent

2-3-4 split

2-3-4 to r-b

rotation-color

Right: split w/3-node parent

2-3-4 split

2-3-4 to r-b

2-3-4 to r-b

color
AVL Tree: Overview

- Balanced binary search tree
- Close to a minimum height bst
- Monitors shape and adjusts when needed
AVL: balance by rotation

Rotations are performed when bst needs to be balanced
There are two types of rotation

- Single (left or right)
- Double (left or right)

- load factor \((h_R - h_L) < 0\) : rotate right
- load factor \((h_R - h_L) > 0\) : rotate left
Single rotation: left

+ LF means right heavy, so rotate left.
* 40's LC becomes 20's RC

LF = +2 after 60 is added
Single rotation : right

- LF means left heavy, so rotate right.
- 40's RC becomes 70's LC.

LF = -2 after 35 is added
Double rotation: left

- LF means left heavy, so rotate right.
- Since 20 now became right heavy, we must first rotate left around 20.
- Then we can rotate right around 40.

LF = -2 after 22 is added
Double rotation : right

- LF = +2 after 46 is added

+ LF means right heavy, so rotate left.
* Since 50 now became left heavy, we must first rotate right around 50.
* Then we can rotate left around 30.
12.2

Hashing
Hashing allows for efficient O(1) access

\[ \text{key} \xrightarrow{\text{hashing function}} \text{index} \]

0

1

\ldots

n-1
A hash function must be easy and fast to compute
It must place items evenly throughout the hash table.
Hash functions use integer arguments
HF #1: Digit selection

**Strategy:** select the 4th and last digit

\[
\begin{align*}
\text{hf}(001364825) &= 35 \\
\text{hf}(113356235) &= 35
\end{align*}
\]

**Conclusion:** they do not evenly distribute items in hash table
Strategy: add all the digits together

\[ hf(001364825) = 0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5 = 29 \]
thus, 0 \( \leq \) hf(k) \( \leq \) 81 and table would have 82 entries

\[ hf(001364825) = 001 + 364 + 825 = 1,190 \]
thus, 0 \( \leq \) hf(k) \( \leq \) 2,997 and table would have 2,998 entries
Strategy: make table size a prime number
(reduces collisions)

\[ hf(001364825) = 001364825 \mod 101 = 12 \]
Text must be converted to an int
ASCII values lead to collisions

Convert:
"NOTE" and "TONE"

Use ASCII values:
N = 78, O = 79, T = 84, E = 69

Add digits:
NOTE = 78 + 79 + 84 + 69 = 310
TONE = 84 + 79 + 78 + 69 = 310

Obviously, this results in collisions
Horner's rule simplifies computations

Convert:
"NOTE"

Use values 1–26 and represent as binary:

N = 14 : 01110
O = 15 : 01111
T = 20 : 10100
E = 5 : 00101

Concatenate:
01110 01111 10100 00101 = 474,757 <= rather large number of entries
Horner's rule simplifies computations

Express in base 32:

\[
\begin{array}{cccc}
01110 & 01111 & 10100 & 00101 \\
32^3 & 32^2 & 32^1 & 32^0
\end{array}
\]

\[
= 474,757 \leq \text{rather large number of entries}
\]

Collect terms:

\[
14 \times 32^3 + 15 \times 32^2 + 20 \times 32^1 + 5 \times 32^0
\]

Apply Horner's rule:

\[
( (14 \times 32 + 15) \times 32 + 20 ) \times 32 + 5
\]

Computations can lead to int overflow, so apply hf to each () expression:

\[
h( h( h( 14 \times 32 + 15 ) \times 32 + 20 ) \times 32 + 5 )
\]
You must resolve collisions
Use one of these two common resolutions

- Open addressing using probing (use another hash table entry)
- Restructuring the hash table
You have three open addressing choices

- Linear probing
- Quadratic probing
- Double hashing
Linear probing is easy

\[ h(key) = key \mod 101 \]

resolution: \( h + i \)

\[
\begin{array}{ll}
22 & 7597 \\
23 & 4567 \\
24 & 0628 \\
25 & 3658 \\
\end{array}
\]

clusters form with linear probing
Quadratic probing eliminates primary clusters

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h + 1^2$</td>
<td>7597</td>
</tr>
<tr>
<td>$h + 2^2$</td>
<td>5467</td>
</tr>
<tr>
<td>$h + 3^2$</td>
<td>0628</td>
</tr>
<tr>
<td></td>
<td>3658</td>
</tr>
</tbody>
</table>

$h = 7597 \mod 101 = 22$

$h(7597) = 22$, $t[22]$
$h(5467) = 22 + 1^2$, $t[23]$
$h(0628) = 22 + 2^2$, $t[26]$
$h(3658) = 22 + 3^2$, $t[31]$

$h(\text{key}) = \text{key mod 101}$

resolution: $h + i^2$
Double hashing drastically reduces clustering

\[ h_1(\text{key}) = \text{key} \mod 11 \]
\[ h_2(\text{key}) = 7 - (\text{key} \mod 7) \] [this is the probe step]

\[ h_1(58) = 3 \]
\[ h_1(14) = 3, \ t[3]: \text{collision} \]
\[ h_2(14) = 7, \ t[10] \]
\[ h_1(91) = 3, \ t[3]: \text{collision} \]
\[ h_2(91) = 7, \ t[10]: \text{collision} \]
\[ (10 + 7(\text{probe})) \mod 11 = 6, \ t[6] \]
You have two restructuring choices

- Buckets
- Separate chaining
Buckets use arrays as hash table elements

- Array of B items
- Hash table
- Size B of array in each element defines collision frequency
Separate chaining uses linked lists instead

This is a better strategy since a linked list can grow easily
Hashing efficiency depends on the load factor

\[ \alpha = \text{number of items} / \text{table size} \]

- As the table fills, \( \alpha \) increases, along with collisions
- As collisions increase, efficiency decreases
- Keep \( \alpha \) around 2/3
Things that make for a good hashing function

- Easy and fast to compute
- Distributes items evenly throughout table
  - random and nonrandom data