

COMP20003 Workshop Week 6

Hashing + Assignment 2

Distribution Counting (aka. Counting Sort)

Hashing

2-3-4 Trees

LAB

- Implementation W6.5 (a small exercise)
- Assignment 2

Distribution Counting: An Unusual Sorting Algorithm

Strengths

Does **not** rely on **key comparisons**

Runs in linear time $\Theta(n + k)$, where k is the key range $\rightarrow \Theta(n)$ time if $k \in O(n)$

Stable (preserves the order of equal keys)

Limitations

Requires $\Theta(n+k)$ **extra memory**

Inefficient when the key range k is much larger than $n \rightarrow$ **not a general-purpose sort**

Constraints

Keys must be **integers** (or *mapped to integers*) **in small range** k

Operation

counts the frequency of each key

builds the cumulative array to determine positions

places elements into the output array in order

Example

Special Example: sort an array where the keys are non-negative integers, each ≤ 2 :

input keys: {0, 1, 2, 0, 0, 1, 2, 1, 1, 0, 0, 0}

freq(0) = 6

keys 0 start
from index
0

freq(1) = 4

keys 1 starts
from index
6

freq(2) = 2

keys 2 starts
from index
10

Output Sorted array: { 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 2 }

Note: here $k = 2$

Array $F[] = \{6, 4, 2\}$ is the frequency array

Array $C[] = \{0, 6, 10\}$ is the cumulative array

Counting Sort for sorting array $A[0..n-1]$

Input: $A[0..n-1]$, k (such that $0 \leq A[i] \leq k$)

Output: $B[0..n-1]$ which is the sorted version of $A[]$

Step 1a: build the *count array* $F[]$ such that

$F[x]$ = frequency of x in $A[]$

Step 1b: transfer $F[]$ to the *cumulative array* $C[]$:

$C[x]$ = starting position of value x
in the sorted array $B[]$.

Step 2: scan $A[]$ from left to right and copy elements A to sorted array $B[]$. For each $A[i]$:

$B[C[A[i]]++] = A[i] \iff \begin{cases} x = A[i]; \\ B[C[x]] = x; \\ C[x]++; \end{cases}$

$A[0..11] = \{2, 0, 1, 0, 3, 0, 1, 2, 1, 1, 0, 0\}$

$k = 3$

$F[]$ = count table

idx 0 1 2 3

5	4	2	1
---	---	---	---

$C[]$ = table of "next position for i "

idx 0 1 2 3

0	5	9	11
---	---	---	----

$B[] =$

--	--	--	--	--	--	--	--	--	--	--	--

$A[] = \{2, 0, 1, 0, 3, 1, 2, 1, 1, 0, 0, 0\}$

Peer Activity: Sorting Numbers

Suppose that we have a sequence of binary numbers (either 0 or 1).

Which sorting algorithm is best suited for sorting this sequence of numbers?

- a. Quick sort
- b. Selection sort
- c. Insertion sort
- d. Distribution counting

Hashing – Key Features

Concept:

Maps keys to indices in an array (**hash table**) using a **hash function**.
Aims for near-constant time insertion, deletion, and search.

Strengths:

Fast **average-case** operations: **$O(1)$** for insert, search, delete.
Flexible: can store integers, strings, or complex objects via suitable hash functions.

But:

$O(n)$ **worst** case for insert, search, delete.

Typical Use Cases:

Implementing dictionaries, sets, symbol tables, caches.
Fast membership testing and lookups.

Example of Hashing

```
// hash function
int hash(int key) {
    return key % 13;
}
```

Want to insert:

14, 20, 33, 46, 31, 72, 16

hash(14) == 1

hash(30) ==

hash(17) ==

Index	0	1	2	3	4	5	6	7	8	9	10	11
Key												

Example of Collision

```
// hash function
int h(int key) {
    return key % 13;
}
```

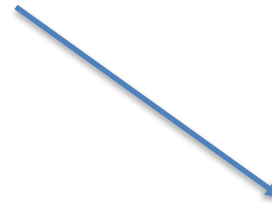
Want to insert:

14, 20, 33, 46, 31, 72, 16

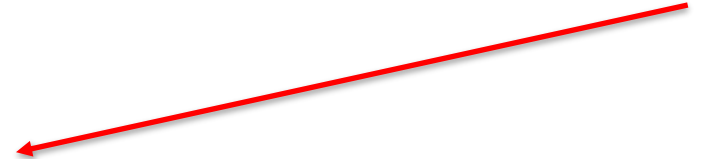
$h(14) == 1$



$h(20) == 7$



$h(33) == 7$



Index	0	1	2	3	4	5	6	7	8	9	10	11
Key		14						20				

Collision at index 7: where to put 33?

Collision Resolution: Method 1A – Open Addressing with Linear Probing

Open Addressing : store all elements in the table itself; on collision, probe for next empty slot.

- **1A - Linear Probing: *probe*** with step=1: $h(k,i) = (h(k) + i) \bmod m$

```
// hash function
int h(int key) {
    return key % 13;
}
```

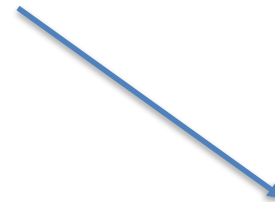
Want to insert:

14, 20, 33, 46, 31, 72, 16

$$h(14) == 1$$



$$h(20) == 7$$



$h(33) == 7$
index 7 is busy
found vacant
location 8 when $i=1$

Index	0	1	2	3	4	5	6	7	8	9	10	11
Key		14						20	33			

Collision Resolution: Method 1A – Open Addressing with Double Hashing

- 1B- Double Hashing: *probe* with step= h2(x)

$$h(k,i) = (h1(k) + i * h2(k)) \bmod m$$

```
// hash function
int h(int key) {
    return key % 13;
}
```

```
// using 2nd hash function
int h2(int key) {
    return key % 5 + 1;
}
```

$h(14) == 1$

$h(20) == 7$

$h(33) == 7$
index 7 is busy
found vacant location
 $7 + 4 = 11$ when $i = 1$



Index	0	1	2	3	4	5	6	7	8	9	10	11
Key		14						20				33

Want to insert: 14, 20, 33, 46, 31, 72, 16

Collision Resolution: Method 2 – Separate Chaining

Separate Chaining : store a linked list (or other dynamic structure) at each hash table slot; all keys that hash to the same index are inserted into that list.

```
// hash function
int h(int key) {
    return key % 13;
}
```

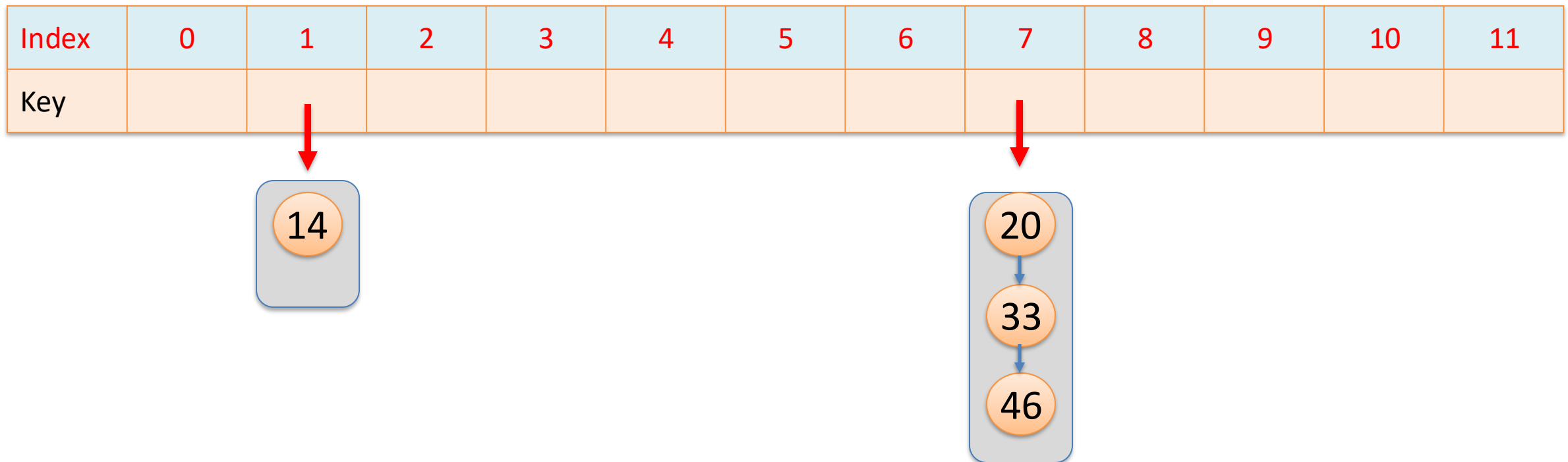
Want to insert:

14, 20, 33, 46, 31, 72, 16

$h(14) == 1$

$h(20) == 7$

$h(33) == 7$



Peer Activity: Overfilled Hash Table

Suppose that we have a **hash table** that:

- uses linear probing/double hashing for collision resolution
- is currently full

What should be done to insert another item into the hash table?

- a. Give up; nothing can be inserted into an overfilled hash table.
- b. `realloc()` the key array and insert the new item into this hash table.
- c. `realloc()` the key array, rehash the existing keys, and insert the new item into this hash table.
- d. Create another identical, empty hash table and insert the new item there.

Peer Activity: Overfilled Hash Table

What should be done to insert another item into the hash table?

- c. `realloc()` the key array, rehash the existing keys, and insert the new item into this hash table.

Why?

- o `realloc()` the key array to get more space to store keys
- o rehash existing keys to redistribute them over the enlarged key array

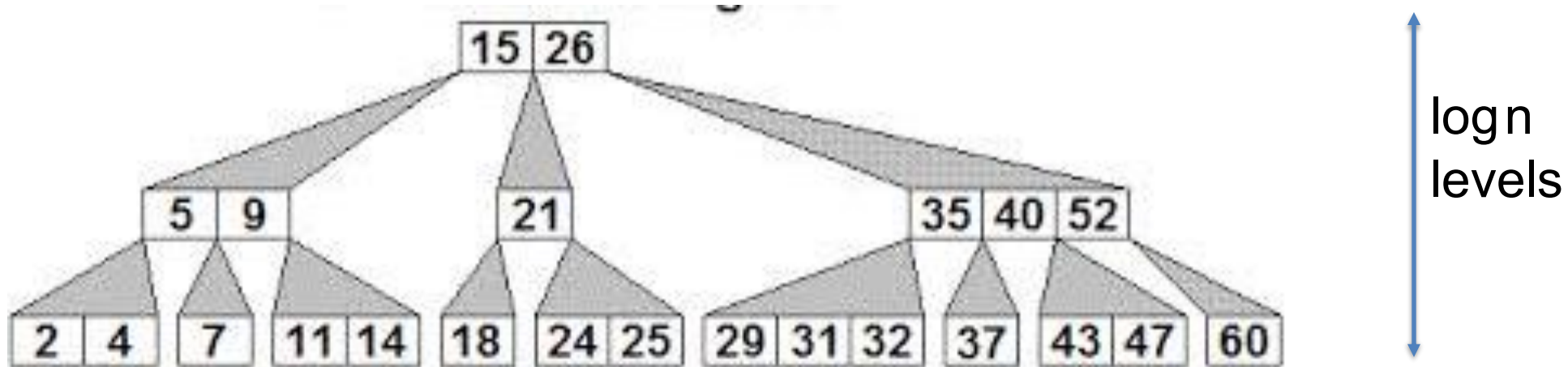
Suppose that we have a **hash table** that:

- o uses linear probing/double hashing for collision resolution
- o is currently full

2-3-4 Trees (B-trees of order 4)

What? It's a search tree, but not a binary tree!

Each node might have 1 to 3 keys, and hence 2 to 4 children.



always balanced: all leaf nodes are in the same level

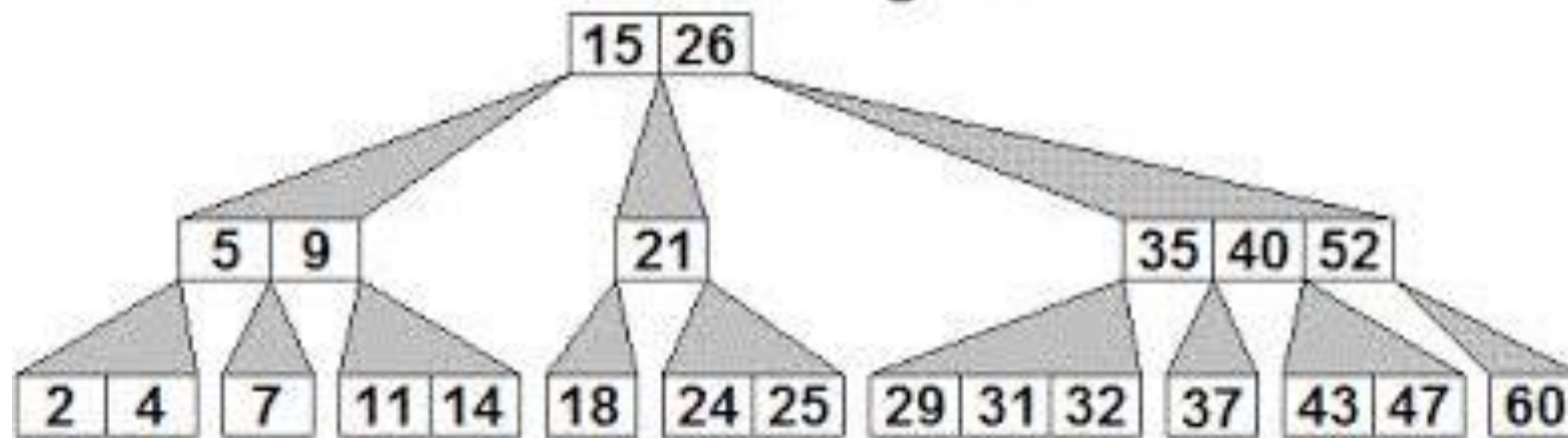
→ the height of the tree is $O(\log n)$

→ search/insert/delete is $O(\log n)$

2-3-4 Trees: Insertion

How to insert a new data **key** :

- start from root, use **key** to go down to a **leaf node**, and **insert key to that leaf node**
- if the **leaf node** has ≤ 2 data: insert to that node
- if the **leaf node** has 3 data: promote the median key to its parent *before insertion*
- the promoting might continue several levels upward until getting a parent with ≤ 3 data



insert 8 is easy: node [7] become [7,8]

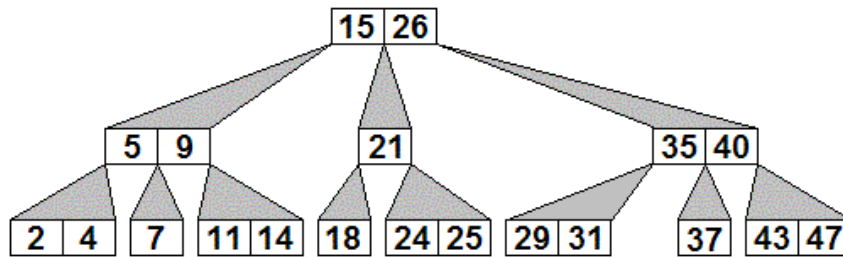
insert 30 :

- promote 31... → **promote 40**
- insert 30 to node 29 → (29,30)
- **promote 40** to the root

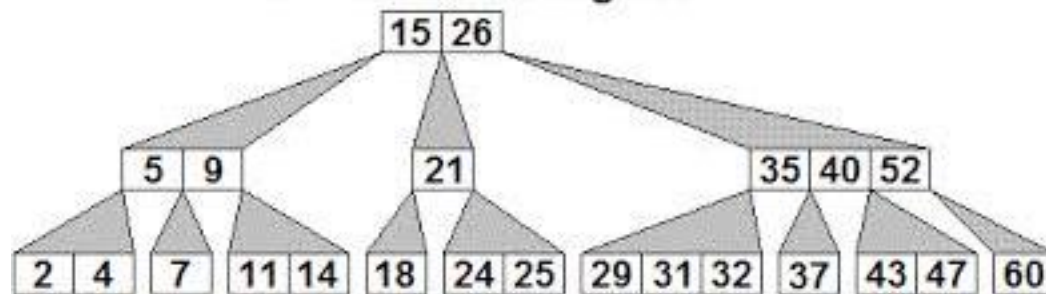
Class example:

Insert the following keys into an initially-empty 2-3-4 Tree.

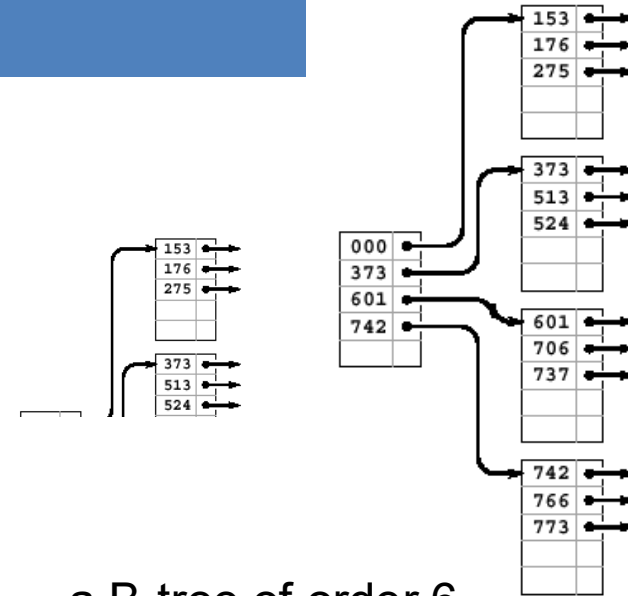
20 10 5 15 30



2-3 trees= B-trees of order 3
(order= max number of children)



2-3-4 trees= B-trees of order 4



a B-tree of order 6

B-tree principles

- Always insert at leaves
- When a node full: promote the median data to the node's parent [and walk up further if needed]

Image sources: ?? and <http://anh.cs.luc.edu/363/notes/06DynamicDataStructures.html>

Practice: W6.2

- Understanding linear probing & double hashing with W6.2
- Programming with W6.3:
 - hash table framework already implemented
 - **just implement 3 functions in hashT.c**

Programming Notes:

- functions `insertLP` and `insertDH5` are used in `insert`,
- the parameter `key` in `insertLP`, `insertDH5`, `insertDH` is actually the first mapping position, `value` is the step (ie. value of `h2(x)`)

Tips:

- start with `insertDH`
- then, `insertLP` and `insertDH5`
- before implementing any function, read its comments in `hashT.h`

Requirements: Code & Report

- Report: perhaps next week
- what to consider when writing the code?

Assignment 2 Strategies

- Choose a good and working version of A1 to start, for example:
 - your code (which is the best!)
 - Anh's solution (not the best, but in the workshop style)
 - Assignment 1 solution (needs more time to digest)
- If the A1 version is good, then it's
 - easy to add just a single module for Patricia trie
 - simple to adapt the main() for using in Assignment2
- Report: understand report's requirements by reading the specs, section "Assessment", then:
 - plan your report to address all 5 key points
 - build the hypothesis
 - plan the experiments to support/reject the hypothesis
 - think about having graphs to compare efficiency

Distribution Counting summary

Unlike other sorting algorithms, *Distribution Counting does not use key comparison.*

Normally not applicable. Can only be useful when

- keys can be considered as **integers in small range**
- ie. when $\min \leq \text{keys} \leq \max$ and $\max - \min \in O(n)$

Time complexity, supposing $r = \max - \min + 1$:

- $P(n+r)$, or
- $P(n)$ if $r \in O(n)$

Special properties:

- *not in-place*, ie. requiring additional arrays for data records
- additional memory: $P(n+r)$, or $P(n)$ if $r \in O(n)$
- the sorting is *stable*, ie. it preserves the relative order of equal keys