

Chapter 2: Searching Algorithms



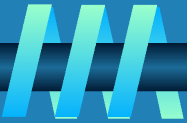
Sequential Search Algorithm and Complexity in arrays

Binary Search Algorithm and Complexity in arrays

Searching in Trees - Balanced Binary Trees

Searching using Hash tables

Sequential Search

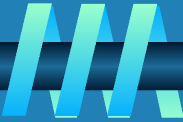


Searching involves looking for a key in an array and returning the index if the key is found.

In sequential search we simply run through the array from beginning to end, looking for the key. As soon as the key is found we quit.



Sequential Search



key=45

5 24 12 45 18 9
↑ i=1

5 24 12 45 18 9
 ↑ i=2

5 24 12 45 18 9
 ↑ i=3

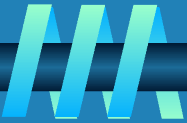
5 24 12 45 18 9
 ↑ i=4

S.Kanchi

SequentialSearch= 4

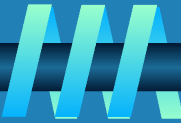


Sequential Search



```
procedure sequentialSearch (A:array,key:integer,  
n:integer):integer  
var found:boolean;  
var i :integer;  
begin  
    i:=1;  
    found := false;  
    while (i <=n and !found ) do  
        if (key = A[i]) then  
            sequentialSearch:= i;  
            found :=true;  
        else  
            i := i +1;  
        endif  
    endwhile  
    if (!found ) then  
        sequentialSearch :=0;  
    endif  
end;
```

Sequential Search Analysis



**In the worst case, the key will not be in the array and the loop will execute n times
the worst case $O(n)$.**

In the best case, the key could be the first element so that time in that case is $\Omega(1)$







So the average case analysis is needed.



Sequential Search Average Case Analysis



The key can be found in any of the positions 1 to n, or key May not be found. The probability is $1/(n+1)$ in each case and the time in that case is given below.

							
	1	2	3	4	n	not f
Prob	$1/(n+1)$	$1/(n+1)$	$1/(n+1)$	$1/(n+1)$	$1/(n+1)$	$1/(n+1)$
Time	1	2	3	4	n	n



Sequential Search Analysis (Average)



Average-case analysis (when target is not always found).

$$A(n) = \left(\frac{1}{n+1} \right) * \left[\left(\sum_{i=1}^n i \right) + n \right]$$

$$A(n) = \left(\frac{1}{n+1} \sum_{i=1}^n i \right) + \left(\frac{1}{n+1} * n \right)$$

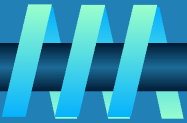
$$A(n) = \left(\frac{1}{n+1} * \frac{n(n+1)}{2} \right) + \frac{n}{n+1}$$

$$A(n) = \frac{n}{2} + \frac{n}{n+1} = \frac{n}{2} + 1 - \frac{1}{n+1}$$

$$A(n) \approx \frac{n+2}{2} \left(\begin{array}{l} \text{As } n \text{ gets very large,} \\ \frac{1}{n+1} \text{ becomes almost 0} \end{array} \right)$$

S. Karachi

Binary Search



First of all binary search works only on sorted array of elements.

Example: 45 78 90 101 234 789 909

We find the middle element in a sorted array and compare it with the key. If the key is found we are done otherwise we make a decision whether we should search left half or right half of the array.

We repeat the same process in the left (or right) half of the array.

This process continues either until element is found or it is determined that the element is not in the array.



Binary Search Example



Binary Search Example: key 72

23 52 72 101 200 305 451 672 921

low=1



23

52

72

101

mid=5



200

305

451

high=9



921

found=false

low=1



23

mid=2



52

72

high=4



101

200

305

451

672

921

found=false

mid=3
low=3



72



binSer=3

high=4



101

200

305

451

672

921

S.Kanchi

found=true

9



Binary Search Example



Binary Search Example: key 451

23 52 72 101 200 305 451 672 921

low=1



23

52

72

101

mid=5



200

305

451

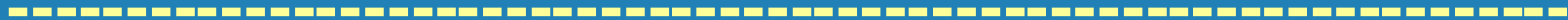
672

high=9



921

found=false



low=6



23

52

72

101

200

305

mid=7



451

672

921

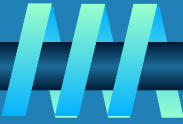


binSer=7

found=true



Binary Search Example



Binary Search Example: key 20

23 52 72 101 200 305 451 672 921

low=1



mid=5



high=9



23 52 72 101 200 305 451 672 921

found=false

low=1



mid=2



high=4



23 52 72 101 200 305 451 672 921

found=false

high=1

low=1



mid=1

23 52 72 101 200 305 451 672 921

found=false

binSer= 0: key value not in list.

S.Kanchi



Binary Search Algorithm (non recursive)

```
procedure binser(A:Array,n:integer,key: integer)
```

```
var found:boolean
```

```
var low, high, mid: integer;
```

```
begin
```

```
    low:=1;
```

```
    high:=n;
```

```
    found := false;
```

```
    while (low <=high && !found) do
```

```
        mid:=(low + high)/2;
```

```
        if (key=A[mid])then
```

```
            found := true;
```

```
        else if (key<A[mid])then
```

```
            high:= mid-1;
```

```
        else
```

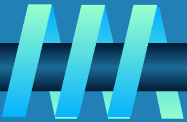
```
            low:= mid+1;
```

```
        endif
```

```
    endif
```

```
endwhile
```

Binary Search Algorithm (nonrecursive)



```
if (found=true) then
    binser:=mid;
else
    binser:=0;
endif
end;
```

How do you find the time complexity of this algorithm?

The best case time is $\Omega(1)$, but the worst case time depends on the number of times the loop executes. So, we rewrite the algorithm using recursion.



Binary Search Algorithm (recursive)

```
procedure recursiveBinser(var key:integer,var low:integer,var  
high: integer):Index;  
begin  
    if (low > high) then  
        recursiveBinser := -1;  
    else  
mid:=(low+high)/2;  
    if (key = A[mid]) then  
        recursiveBinser := mid;  
    else  
        if (key < A[mid]) then  
            high := mid - 1;  
            recursiveBinser := recursiveBinser(low,high,key);  
        else  
            low := mid + 1;  
            recursiveBinser:=recursiveBinser(low,high, key);  
        endif  
    endif  
endif  
end;
```

Binary Search Algorithm (recursive)



```
procedure recursiveBinarySearch(A:Array, key:integer;  
n:integer):integer;  
  
var result:integer;  
  
begin  
    result :=recursiveBinser(key,1,n);  
  
end;
```

The best case time is $\Omega(1)$. The worst case time will be computed using recursion.



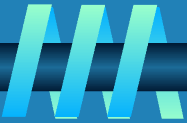
Binary Search Algorithm Analysis

Recurrence Relation:

$$T(n) = 1 + T\left(\frac{n}{2}\right), n \geq 2$$

$$T(1) = 1$$

Binary Search Algorithm Analysis



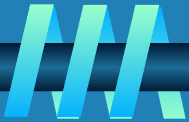
This can be solved using back substitution as follows:

$$\begin{aligned}T(n) &= 1 + T\left(\frac{n}{2}\right) \\&= 1 + 1 + T\left(\frac{n}{4}\right) \\&= 2 + T\left(\frac{n}{2^2}\right) \\&= 3 + T\left(\frac{n}{2^3}\right) \\&= 4 + T\left(\frac{n}{2^4}\right) \\&\vdots\end{aligned}$$

S.Kanchi



Binary Search Algorithm Analysis



The time complexity of binary search is

$$T(n) = \log_2 n + T\left(\frac{n}{2^{\log_2 n}}\right)$$

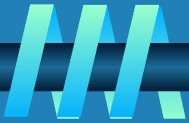
$$= \log_2 n + T(1)$$

$$= \log_2 n + 1$$

$$T(n) = O(\log_2 n)$$



Balanced Trees



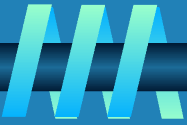
Balanced trees: The problem with BST is that even though the search time is $O(h)$, h can be as big as n .

If we can keep the tree “balanced”, as we perform insert and delete, may be we can achieve a search time of $O(h)$ which would be same as $O(\log n)$.

2-3 trees are one form of balanced binary trees. There are other balanced trees for example, Red-Black Trees, AVL trees etc



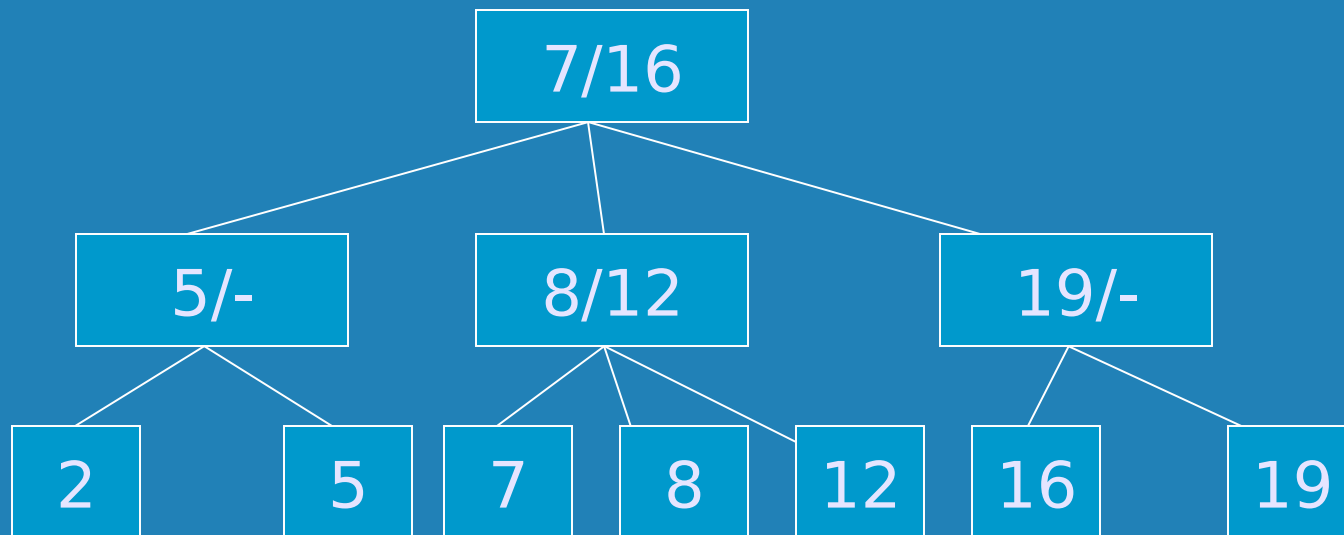
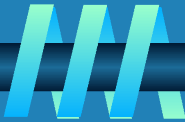
Definition of 2-3 trees:



1. It is a tree in which every non-leaf node has exactly 2 or 3 children. The children are called first child, second child and third child.
2. The length of the path from the root to every leaf is the same.
3. Data is stored only at the leaf.
4. The interior nodes contains two values x and y , x is minimum data in the sub-tree whose root is its second child, and y is minimum data in the sub-tree whose root is the third child, if third child exists.
5. The values at the leaf nodes are always maintained in ascending order when viewed from left to right.

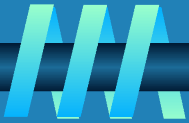


Example of 2-3 trees:



The leaf nodes store the data. Note that the data is in ascending order. The interior nodes store x/y , where x is the minimum of the sub-tree rooted at second child, and y is the minimum of the sub-tree rooted at the third child. If third child does not exist y is left blank.

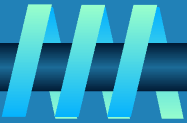
Properties of 2-3 trees



1. If the 2-3 tree has k levels then it has 2^{k-1} to 3^{k-1} leaves
2. If the 2-3 tree has k levels then it has 2^k to 3^k nodes.
3. If the 2-3 tree has n nodes then it has between $\log_3(n)$ to $\log_2(n)$ levels.
4. How much space are we wasting by not storing data at the internal nodes?



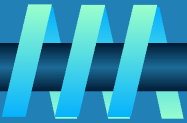
Data structure for 2-3 tree.



```
type DataType = integer;
type 23Node = record
    data:DataType;
    x    :DataType;
    y    :DataType;
    fc   :23node;
    sc   :23node;
    tc   :23node;
    p    :23node;
end;
var root:23node;
```



Basic Operations

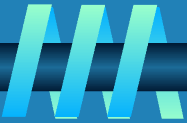


Assume the `curr.data()` gives the data at `curr` node

Similarly assume that `curr.leftChild()`, `curr.RightChild()`, `curr.isLeaf()`, `curr.parent()`, `curr.xValue()` and `curr.yValue()` are all defined.



Advanced Operations



We would like the following operations in $O(\log n)$ or $O(h)$ time.

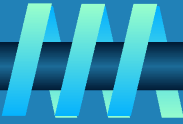
search for key: returns a pointer to the leaf containing key or nil.

insert a key into 2-3 tree: inserts the key in such a way that the new tree is also a 2-3 tree.

delete a key from 2-3 tree: deletes the key in such a way that the new tree is also a 2-3 tree.



Search in 2-3 Tree



Let us look at search. Suppose we want to search for a key:
At each node we encounter, we perform the following check:

if $key < x$ then the key may be found in the sub-tree rooted at
first child of node

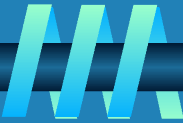
if $x \leq key < y$ then key may be found in the sub-tree rooted at
the second child of node.

if $key \geq y$ then key may be found in the sub-tree rooted at
the third child of the node.

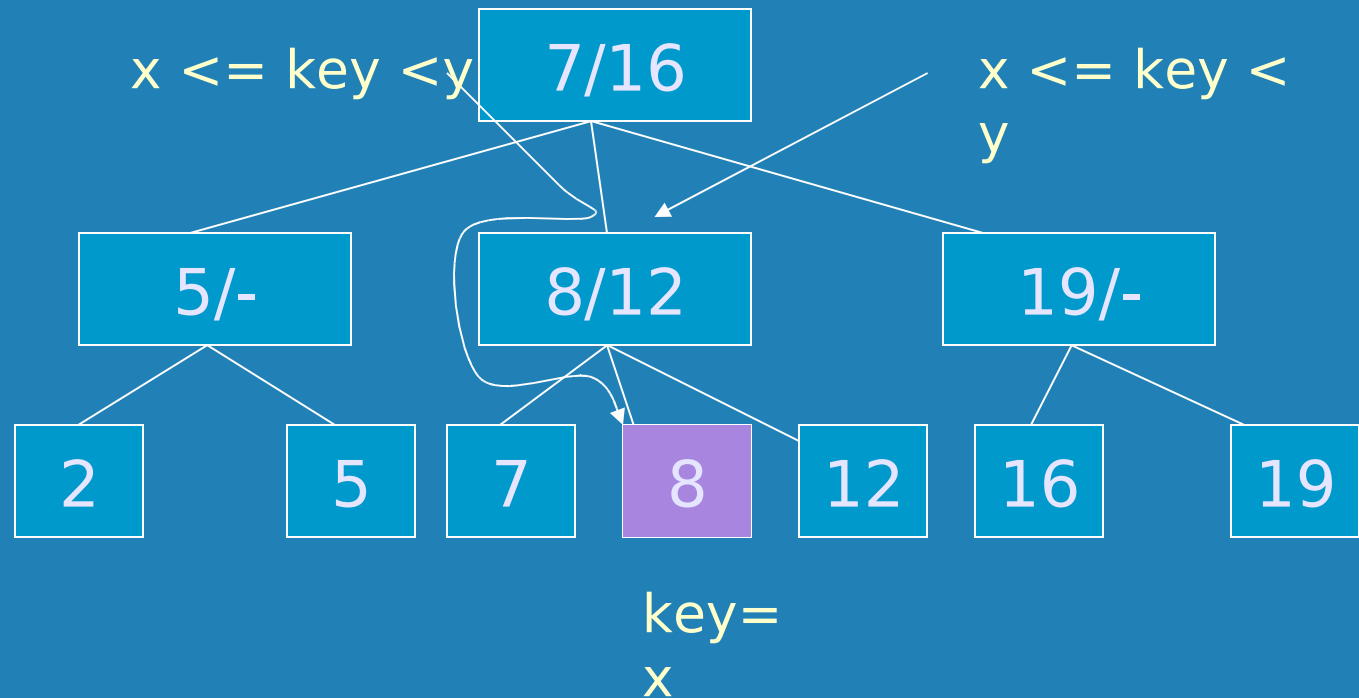
What if there is no third child?



Search in a 2-3 tree



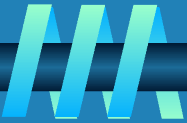
Example of search: Search for 8 in the following tree (key=8).



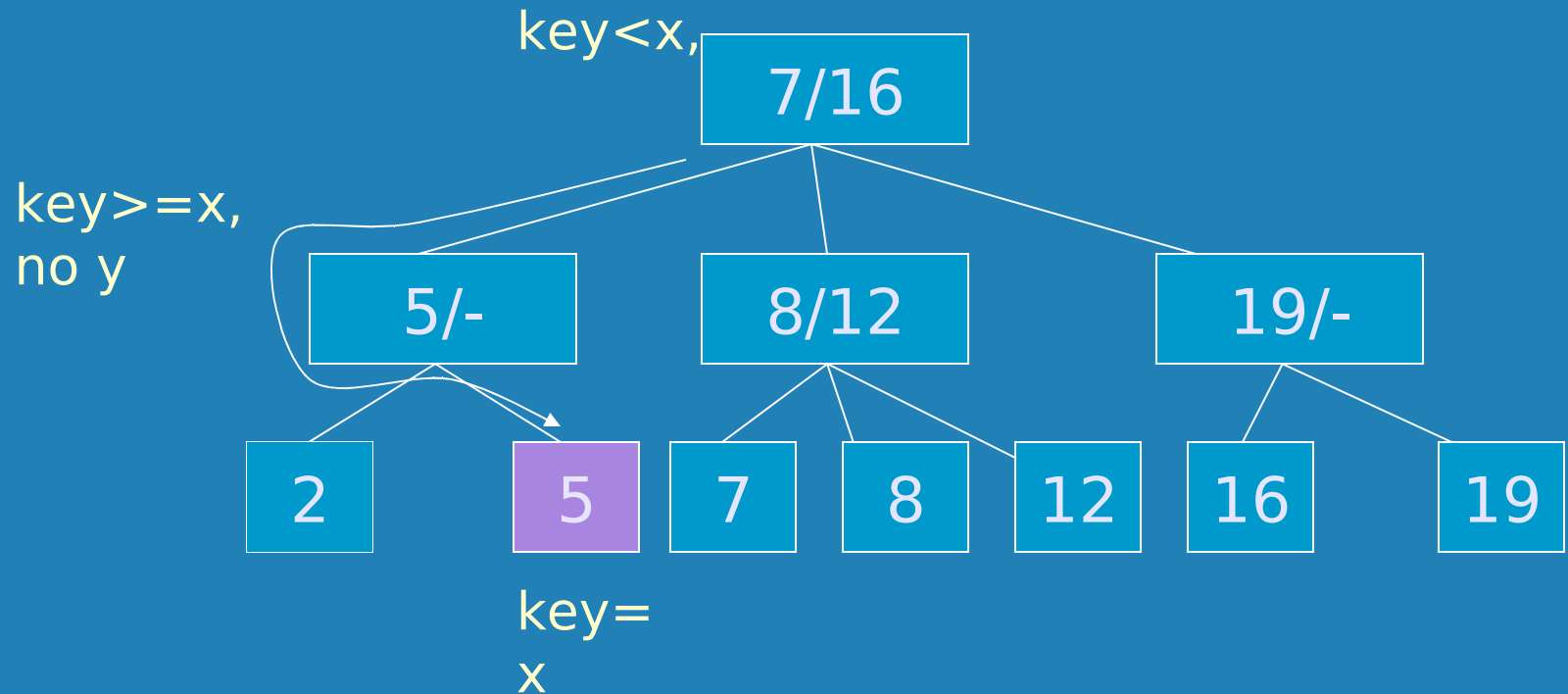
What is the time complexity?



Search in 2-3 Tree

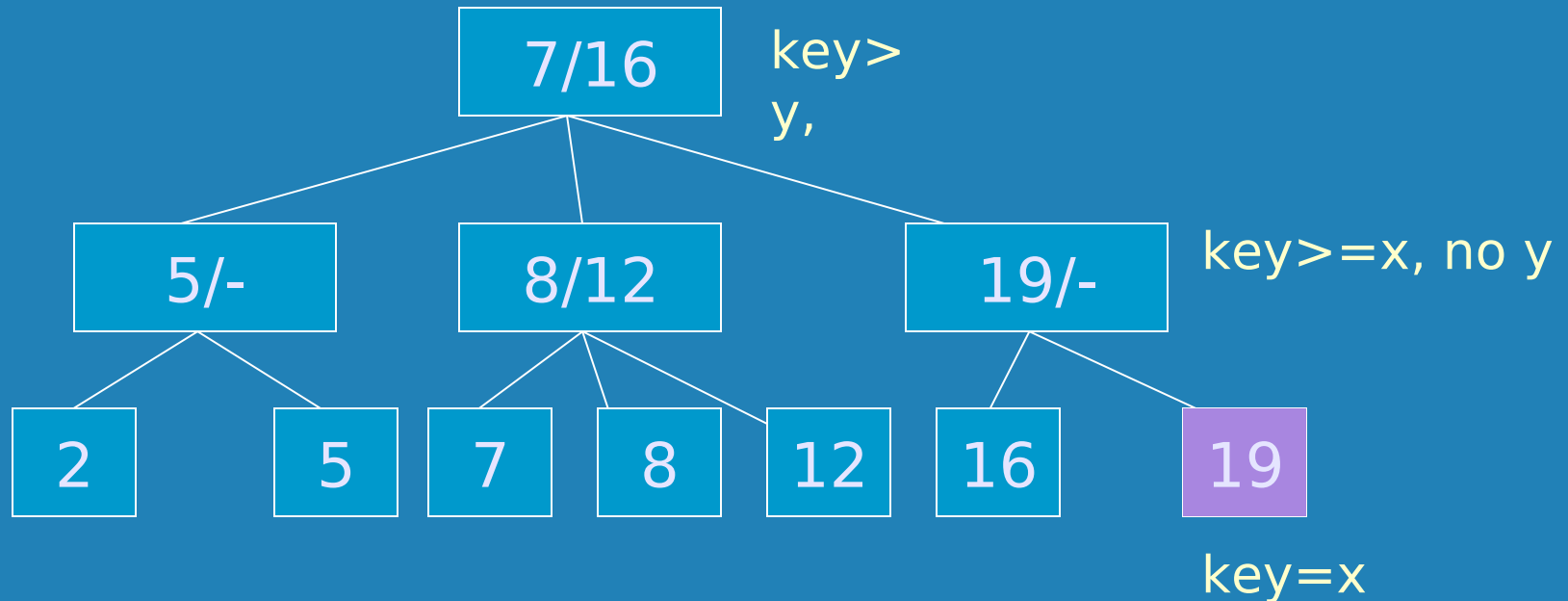


Example of search: Search for 5 in the following tree (key=5).



Search in 2-3 Tree

Example of search: Search for 19 in the following tree (key=19).

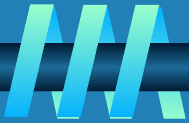


Search in 2-3 tree

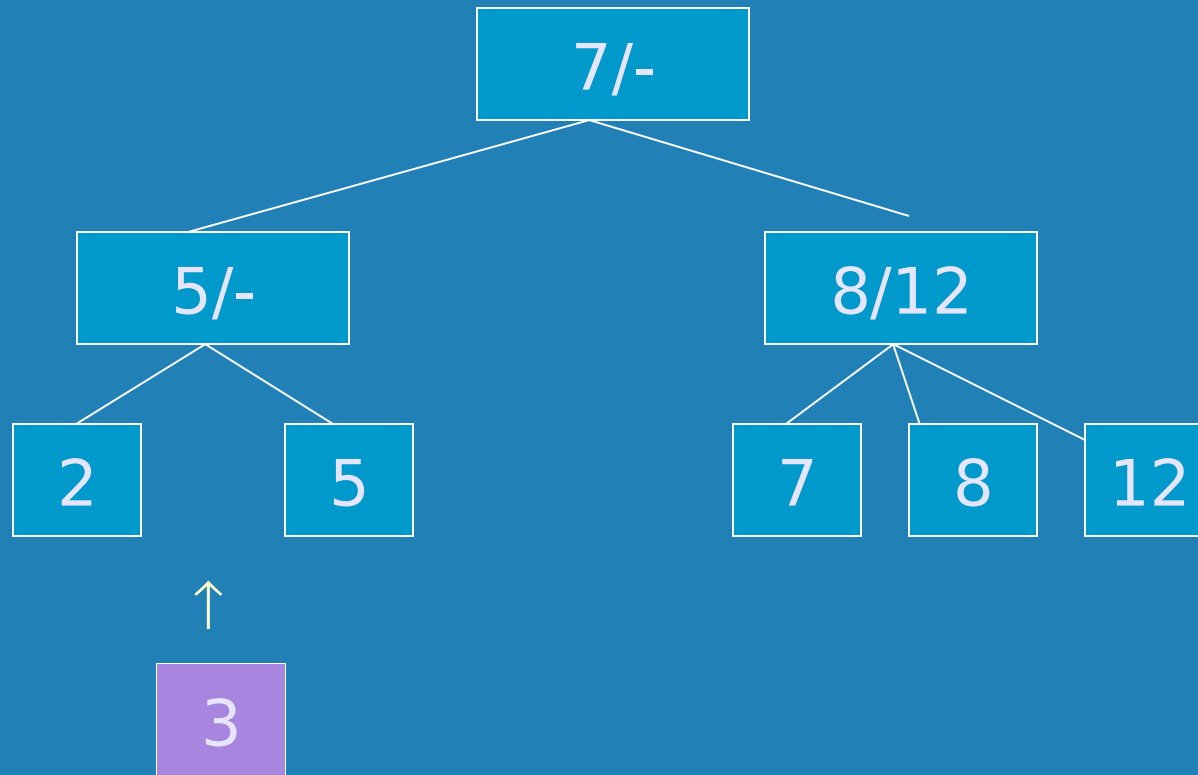


```
procedure search23Tree(root:23node; key:DataType):23node;  
var found :boolean;  
var curr:23node;  
begin  
    found := false;  
    curr := root;  
    while (curr <> nil and found = false) do begin  
        case 1: (curr.isLeaf() and curr.data() = key))  
            found := true;  
        case 2: (key < curr.xValue())  
            curr:= curr.firstChild();  
        case 3: (key >= curr.xValue() and (curr.thirdChild() = nil))  
            curr := curr.secondChild();  
        case 4: (curr.thirdchild <> nil and x <= key <curr.yValue)  
            curr:= curr.secondChild();  
        case 5: (curr.thirdChild() <> nil and key >= curr.yValue())  
            curr:= curr.thirdChild();  
    endwhile;  
    return(curr);  
end;
```

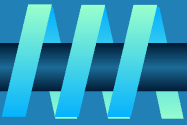
Insert into 2-3 Tree Ex. 1



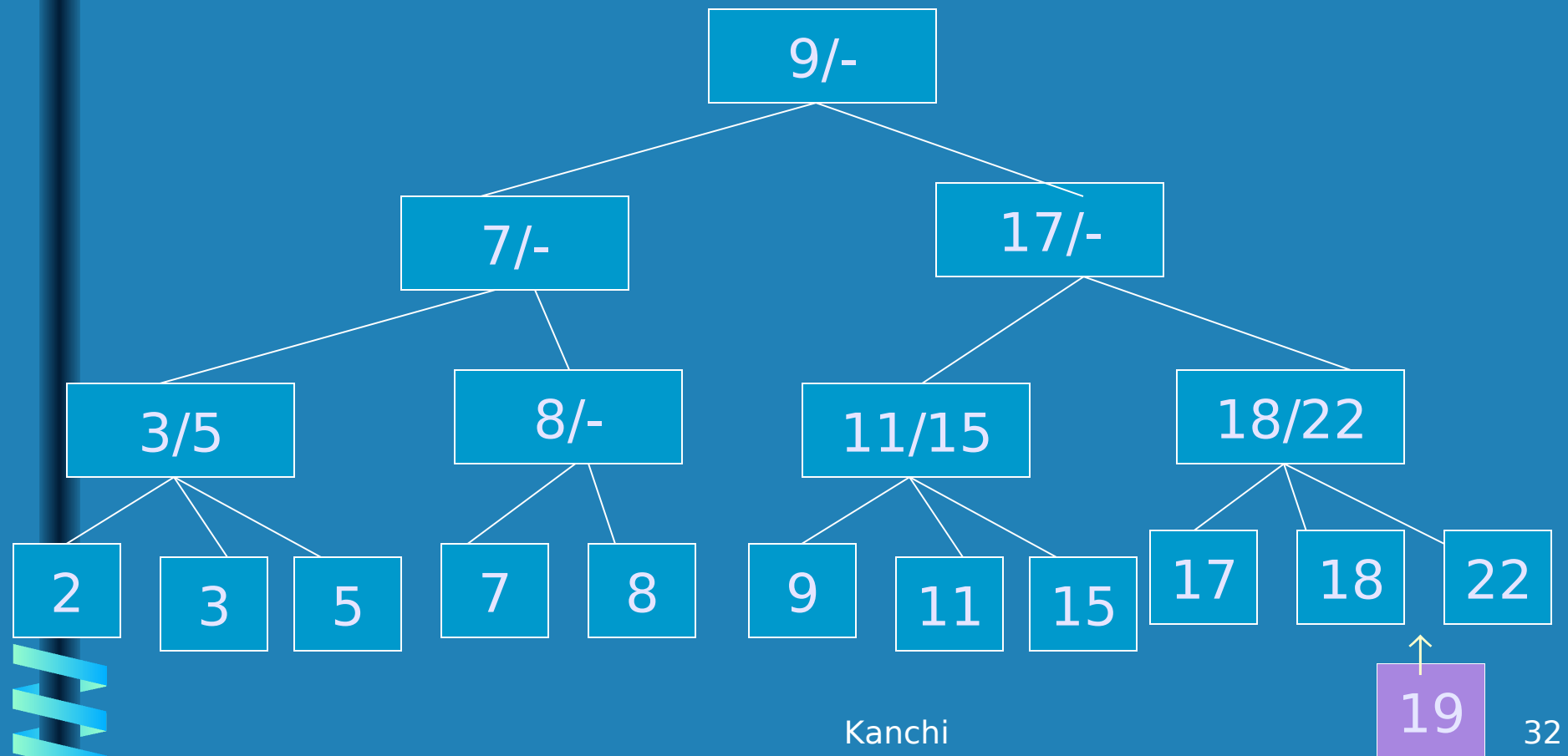
When we want to insert a key, we first search for it, then insert it at the location where it was supposed to be. For example, inserting 3 in the tree below will simply put the node with 3 between 2 and 5.



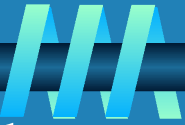
Insert into 2-3 Tree Ex: 2



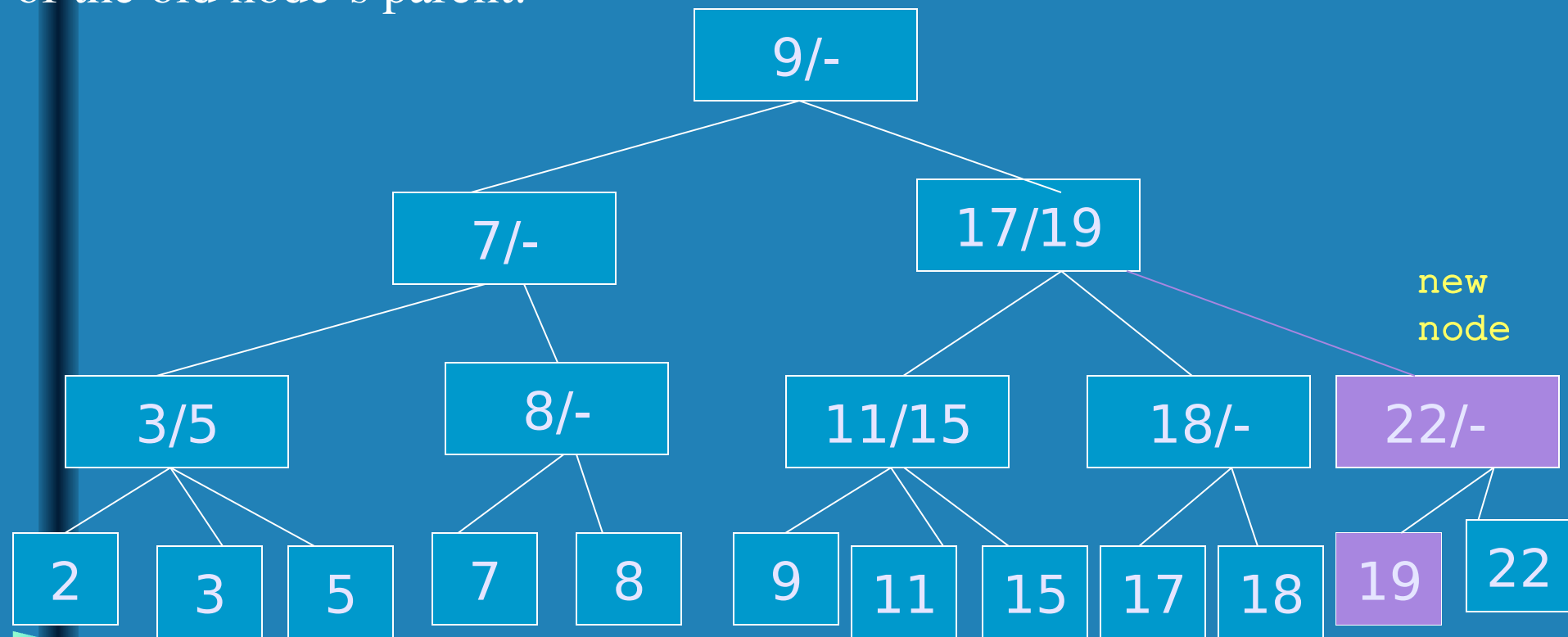
Suppose we wanted to insert 19 into tree below, we have a problem, since the node 18/22 cannot have four children.



Insert into 2-3 Tree Ex: 2



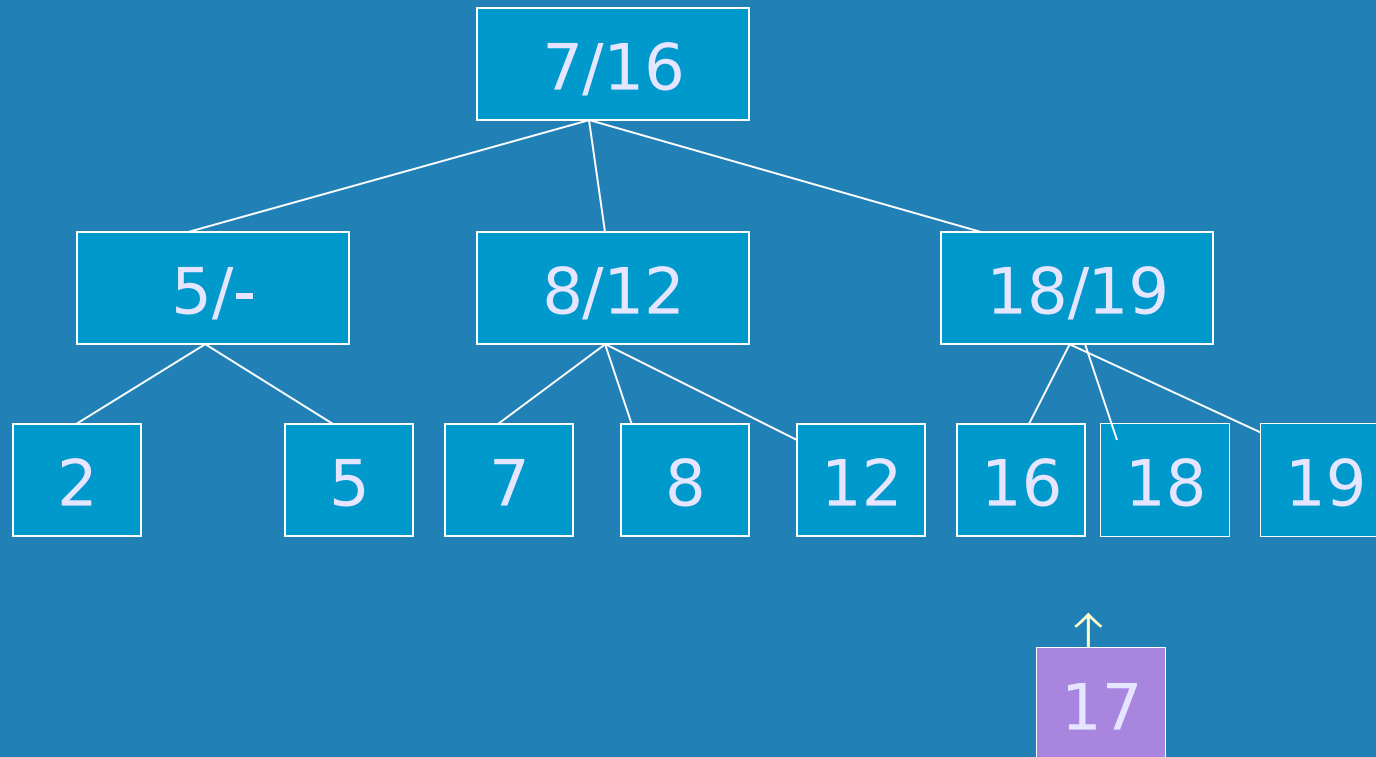
So we split the node 18/22 into two nodes, give two children to old node and give two children to new node and make the new node child of the old node's parent.



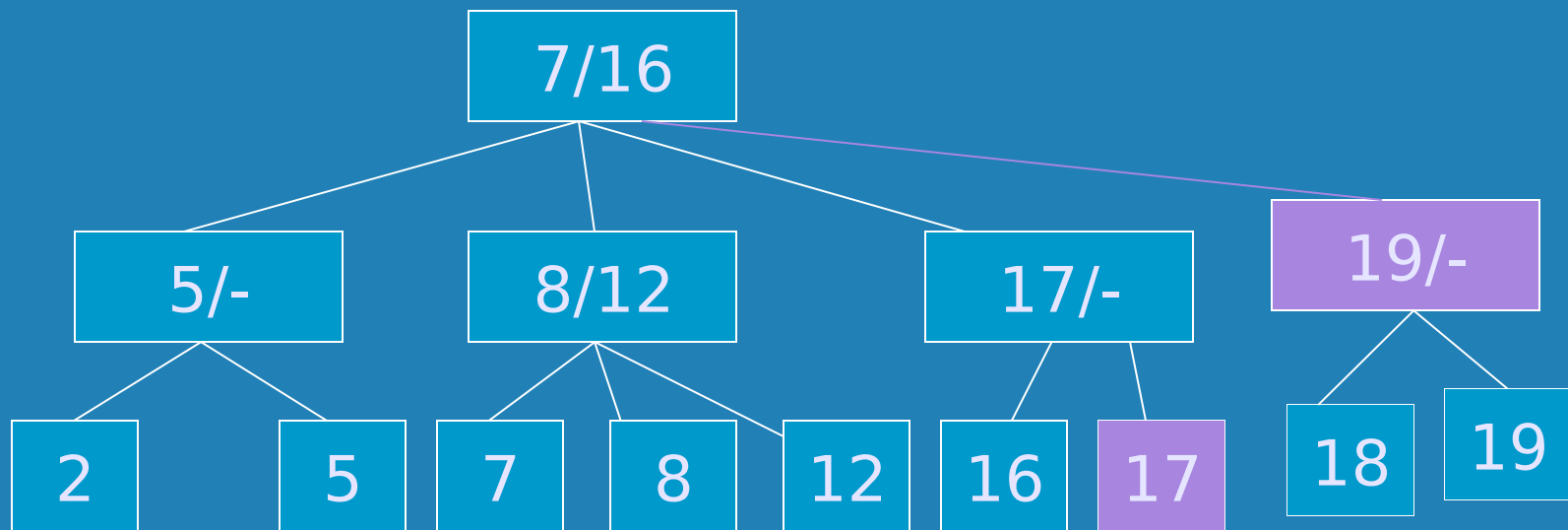
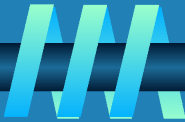
Insert into 2-3 tree Ex: 3



Let us insert 17 into the following tree and still try to maintain a 2-3 tree. We will search for 17, and when we find the right location for 17.



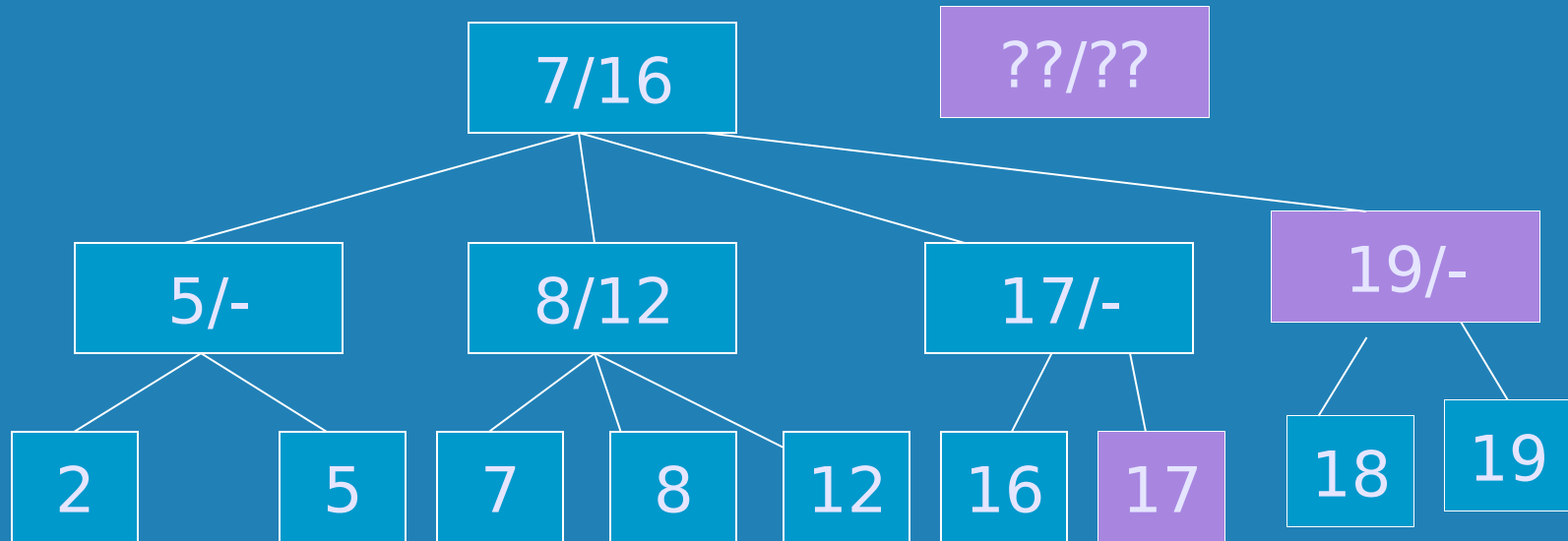
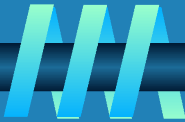
Insert into 2-3 tree Ex: 3



So we will split the node with 19/- into two nodes. Assign two children to old node and two children to the new node. Try to connect the new node to the parent of the old node (root)



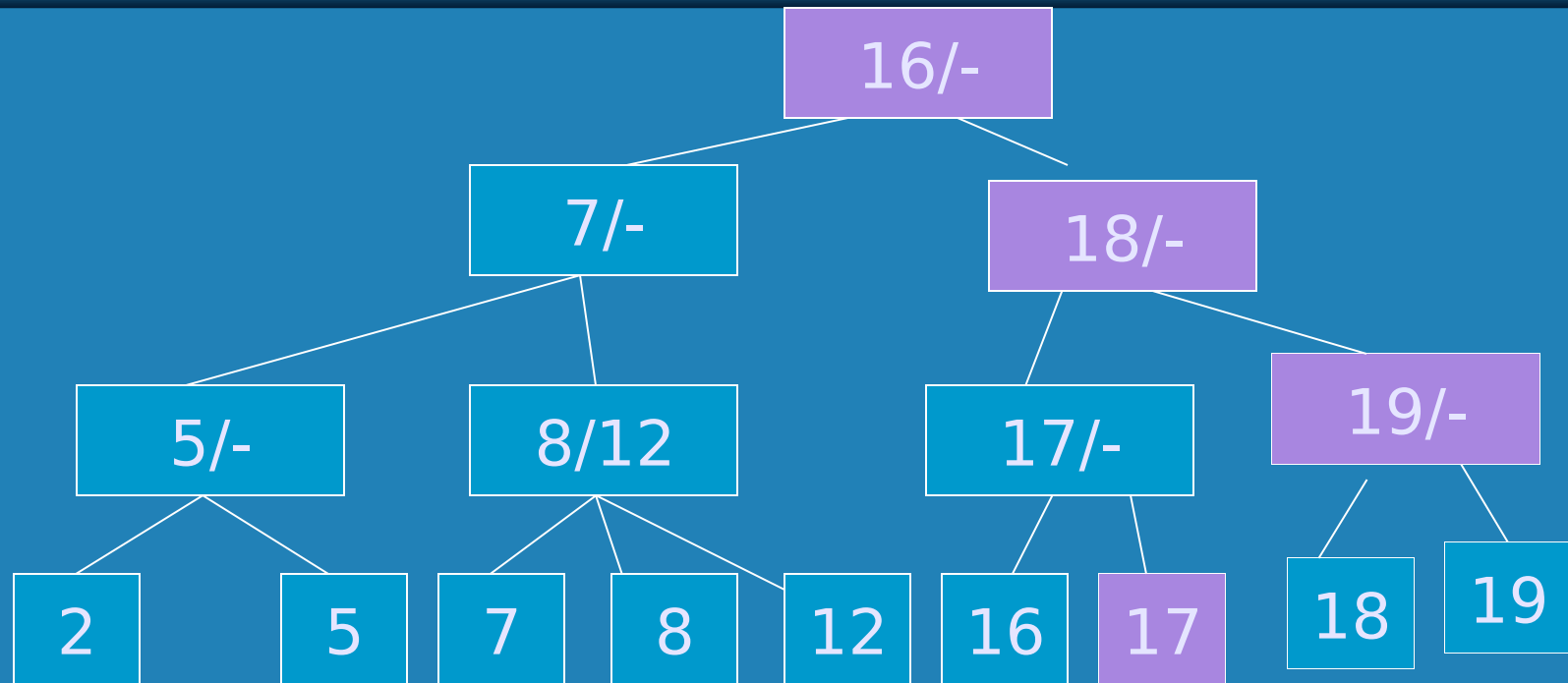
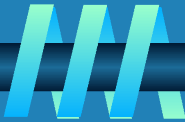
Insert into 2-3 tree Ex: 3



But now the parent has 4 children.. So we split the parent again, and this process continues until either there is a node with 2 children and the new node is added to it, or the root splits.

Kanchi

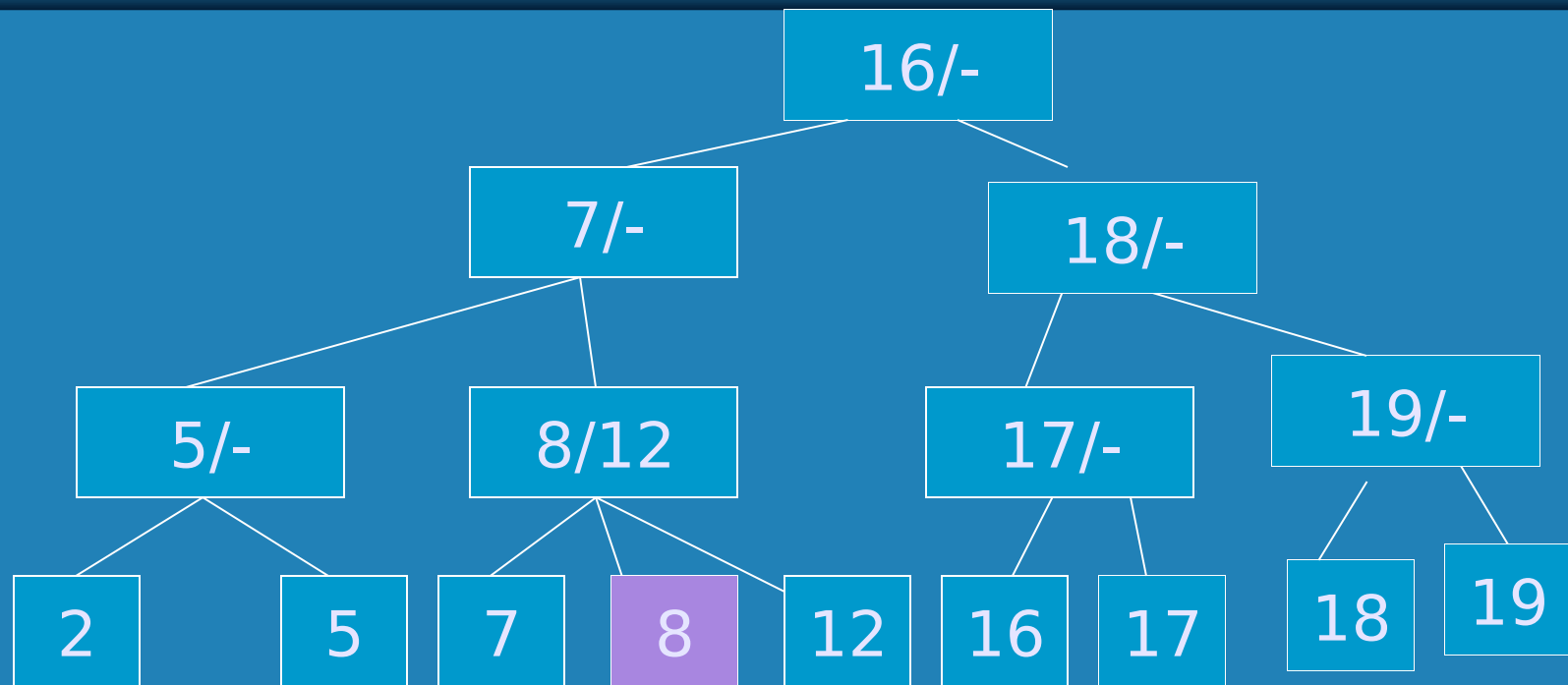
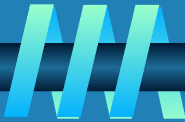
Insert into 2-3 tree Ex: 3



What is the time complexity?

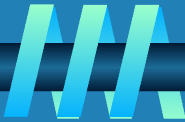


Delete From 2-3 tree Ex: 1

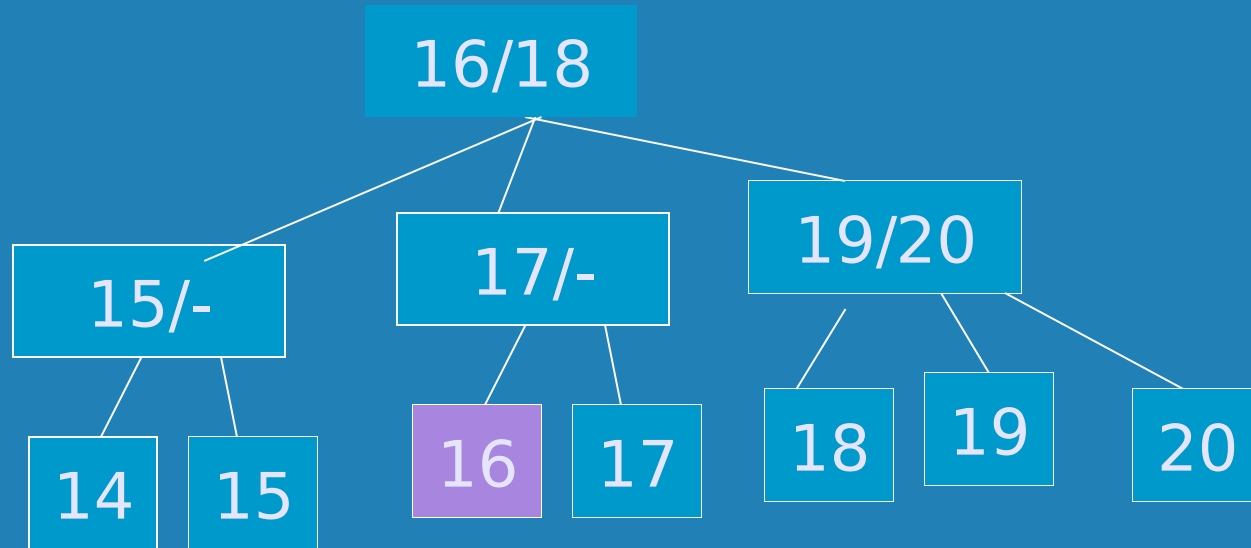


When we want to delete a node, we first search for the key, then we simply remove the leaf, provided it leaves the parent with 2 children. If the parent is left with one child, then we have a problem. For example, Deleting 8 from the above tree is not an issue.

Delete from 2-3 tree Ex:2



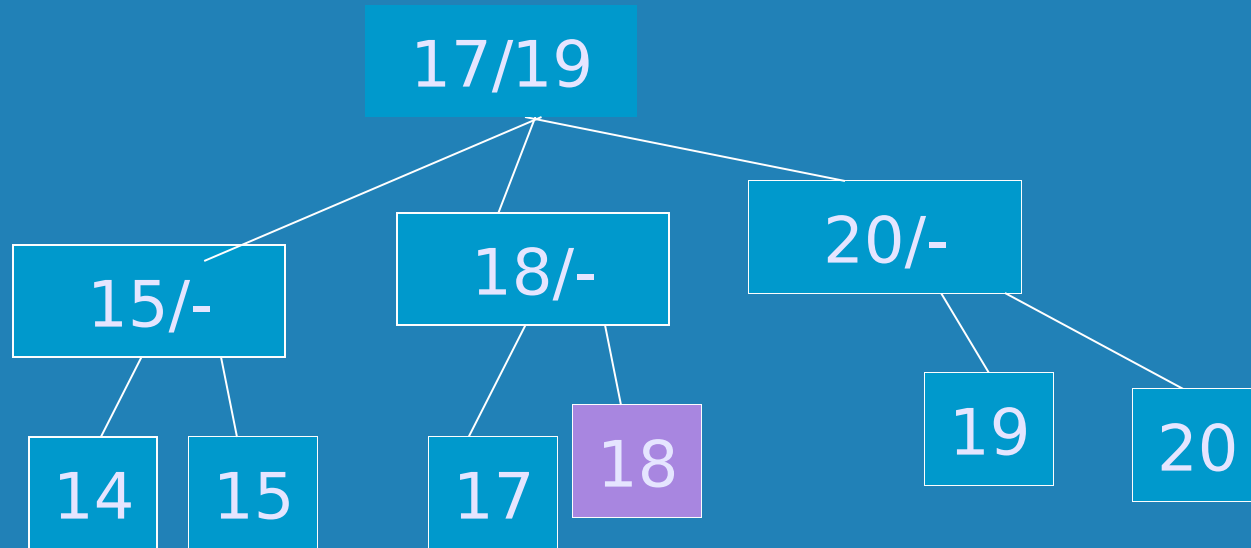
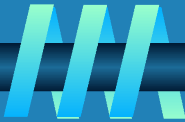
First borrow, if that does not work, then give away



When we want to delete the node with 16 we have a problem since 17/- will be left with one child. Then we find the sibling of 17/- that has three children and assign one child of the sibling to 17/-.



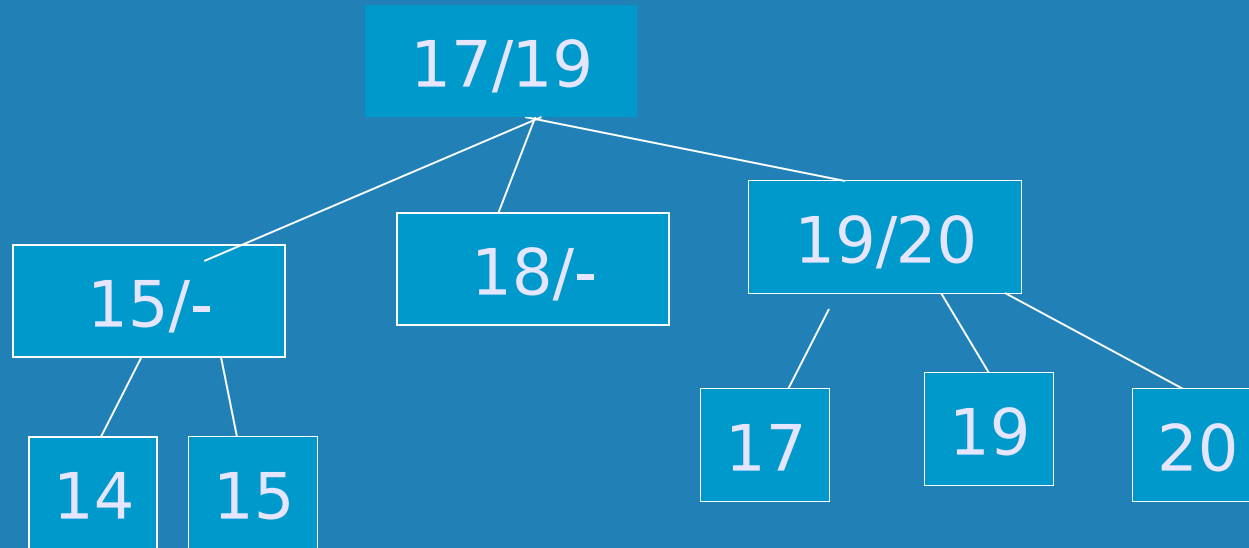
Delete from 2-3 tree Ex: 3



But what if wanted to delete 18 now. 18/- does not have a sibling with three children? We cannot move any children to the 18/-.

We then “merge” or “give away” 18/- with one its siblings (note that now the siblings have only two children) and assign all three children to the new node.

Delete from 2-3 tree Ex: 3

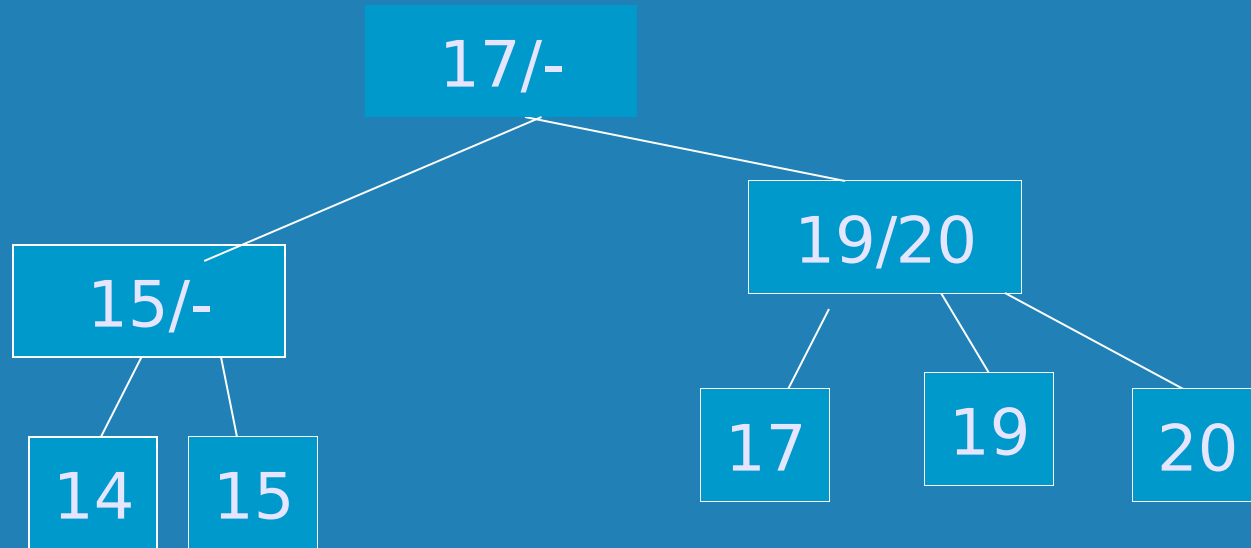
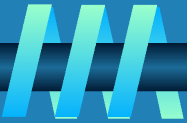


Then we have to delete 18/-. We use the same idea to delete 18/- . If the parent has 3 children then there is no problem, other wise we have to either borrow or “give away”

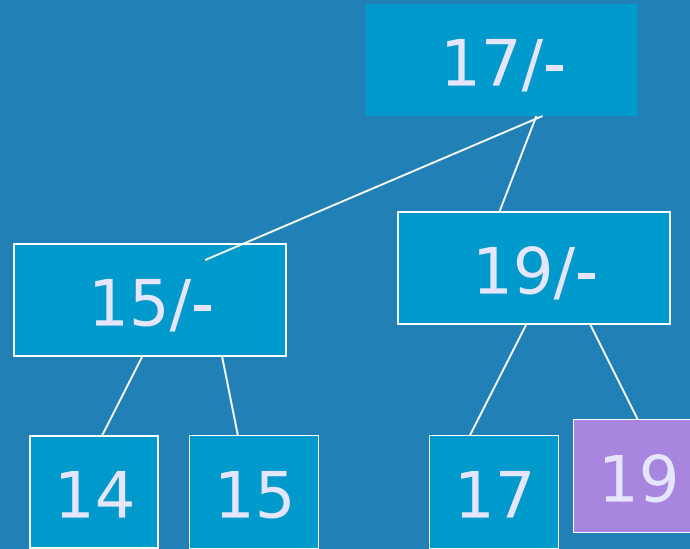
Kanchi



Delete from 2-3 tree Ex: 3

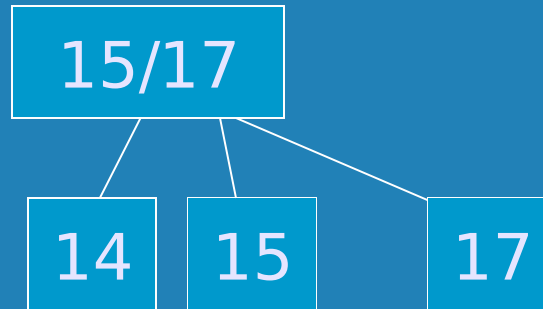
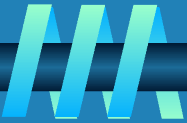


Delete from 2-3 tree Ex: 4



But what if wanted to delete 19 now. 19's parent does not have a sibling with three children? When we “merge” 19/- with one its siblings(note that there is only one sibling) and assign all three children to the new node, we cannot delete 19/- since the parent will have only one child. So the problem may propagate until we reach root. We then delete the root.

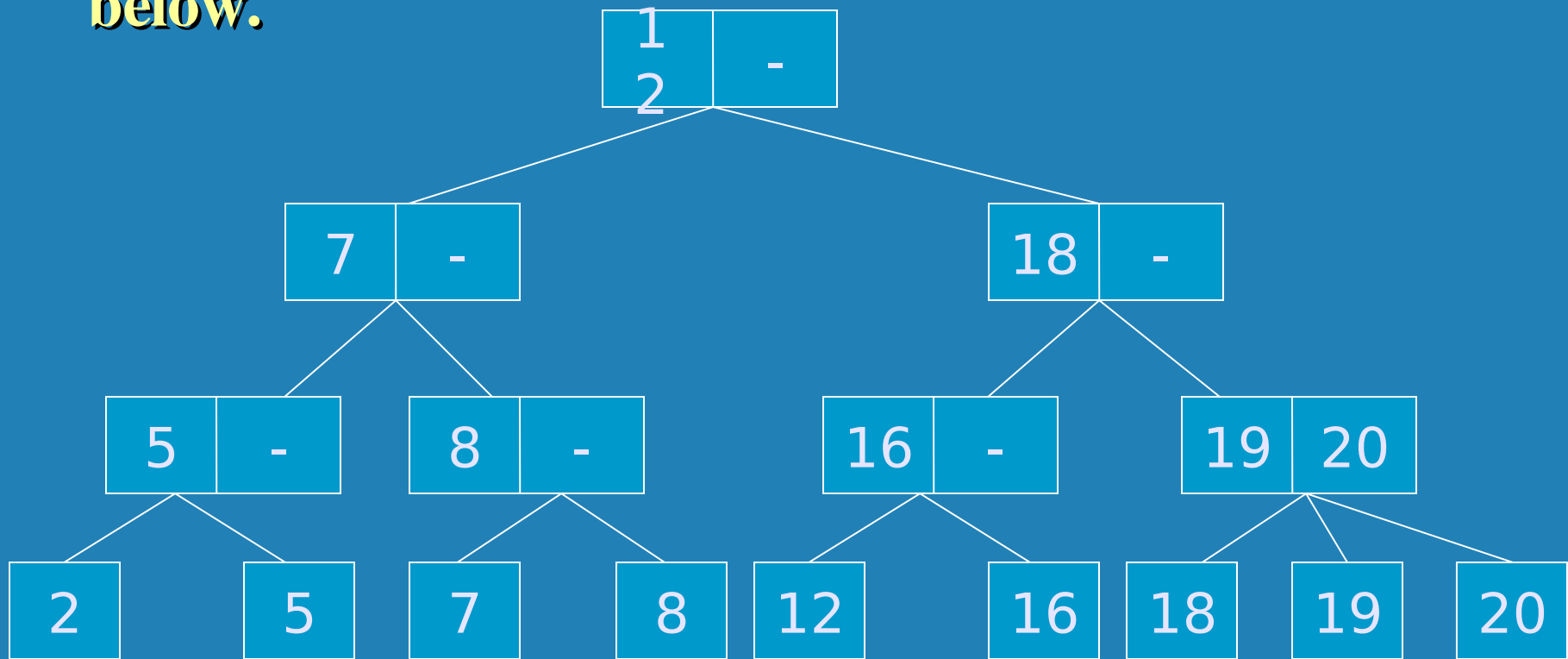
Delete from 2-3 tree Ex: 4



Delete from 2-3 tree- Exercise



Delete the nodes 12, then 18, then 5 from the tree below.



Hashing

Searching for a key in an array takes $O(\log n)$ time using Binary Search if the array is sorted.

Searching in a BST takes $O(h)$ time, and $O(h)$ is $O(\log n)$ if the tree is balanced.

Now we want to get more ambitious, and be able to search in $O(1)$ time!

Given a key we want to be able to locate the key in $O(1)$ time!

How should we store the key so that this can be done?

Hash Tables

We will store the keys in a table called hash-table indexed by the key.

The idea is to compute a procedure called “hash procedure” which computes the index in the table, where the key is to be stored.

For example if the hash procedure h , is the number of letters in the key,

$h(\text{“Larry”}) = 5$, so Larry’s record is stored at location 5.

$h(\text{“Tom”}) = 3$, so Tom’s record is stored at location 3.

If we wanted to locate a record, we simply find the hash procedure of the name, and go to that index.

What is the problem with this technique?

Collision

There is an issue of *collision* since $h(\text{“Tom”}) = h(\text{“Joe”})$. How we store Tom’s record and Joe’s record in the same location?

A hash procedure is called a *perfect hash procedure* if it maps different key’s to different numbers.

Are there perfect hash procedures?

Given less than perfect hash procedures, how do we resolve collision?

Can we still search in $O(1)$ time?

Division Hashing

One of the properties of hash procedure is to make sure that the index it creates is a valid index in the table.

Generally the table size $tsize$ is chosen to be a prime number, and then the hash procedure $h(key) = key \text{ MOD } size$

The key is assumed to be a number, but this can always be arranged for instance you can take a key that is a string and create a number out of it using the number of characters in the string, or using the number of vowels or some other count in the string.

Does each key generate a unique number. NO. There is possibility of collision.

But this is a commonly used hash procedure.

Mid Square procedure

In this technique, the key is squared and mid-part of the result is used as the index to store the key.

For example, to store 3121, we compute $3121^2 = 9740641$.

Assuming that the table size is 1000, we could use the mid-part, 406, as the location to store 3121.

If the key is a string, we first need to convert it to a numeric value using Unicode values of the characters in the string.

Collision Resolution

Although the techniques we have seen so far, appears to search for a key in time $O(1)$, there is an issue with storing the keys when they have the same hash value.

Since the hash procedures of two different keys can be the same, how do we store the keys that hash to the same value. Then, how do we search for a key whose hash value corresponds to more than one key.

To resolve collisions such as above, we use a technique called probing.

Linear Probing

When an element K is to be inserted into the table, first the hash procedure $h(K)$ is computed on the key to find the address. If the address is occupied, then the next available address is found, among the following list:

$h(K)$, $h(K) + p(1)$, $h(K) + p(2)$, $h(K) + p(3)$

The procedure $p(i)$ is called the probing procedure. If the simplest probing procedure $p(i) = i$, is used, the hashing technique is called *linear probing*.

That is, in linear probing, if $h(K)$ is occupied, we look for available address in the following order:

$h(K) + 1$, $h(K) + 2$, $h(K) + 3$

Linear Probing

Arrivals of elements occurs in the following order:

$A_5, A_2, A_3,$

$B_5, A_9, B_2,$

$B_9, C_2,$

1			
2	A_2	A_2	A_2
3	A_3	A_3	A_3
4		B_2	B_2
5	A_5	A_5	A_5
6		B_5	B_5
7			C_2
8			
9		A_9	A_9
10			B_9

Quadratic Probing

The problem with linear probing is that there is ‘clustering’. Once there is a collision, the elements tend to cluster around the same locations. This causes table to be dense in some areas and sparse in others and this makes the search time longer.

If we use a quadratic procedure for $p(i)$, there will not be as much clustering. Let us use the probe sequence as

$h(K)$, $h(K) + 1^2$, $h(K) - 1^2$, $h(K) + 2^2$, $h(K) - 2^2$, $h(K) + 3^2$, $h(K) - 3^2$
etc.

Using the quadratic probing, the placement of the elements in the previous example would be as shown below.

Quadratic Probing

Arrivals of elements occurs in the following order:

$A_5, A_2, A_3,$

$B_5, A_9, B_2,$

$B_9, C_2,$

1	
2	A_2
3	A_3
4	
5	A_5
6	
7	
8	
9	
10	

B_2
A_2
A_3
A_5
B_5
A_9

B_2
A_2
A_3
A_5
B_5
C_2
A_9
B_9

Search by Probing

When searching for a key, we simply rehash the key and then search for key using the same probing sequence as given in linear (or quadratic) hashing.

If we looking for C_2 in the linear probing example above, we will first hit A_2 , since there is no match, we will then look at the probe sequence and find A_3 , B_2 , A_5 , B_5 and then find C_2 .

When the key is in the table, it is found in one of the locations in the hash sequence. If the key is not in the table, we have to travel the hash sequence until make one complete cycle on the hash table.

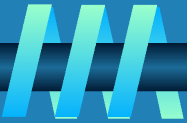
Search Times

Here is the time complexity of search using hashing.

$$LF = \frac{\text{(number of entries in the table)}}{\text{(size of the table)}}$$

Type of search	Linear Probing	Quadratic probing
Successful search	$\frac{1}{2} \left(1 + \frac{1}{1-LF} \right)$	$1 - \ln(1-LF) - \frac{LF}{2}$
Unsuccessful search	$\frac{1}{2} \left(1 + \frac{1}{(1-LF)^2} \right)$	$\frac{1}{1-LF} - LF - \ln(1-LF)$

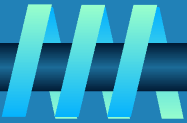
Exercises



1. Write out an algorithm to find x in an ordered list by comparing x to every fourth element until x is found or an entry larger than x is found, and in the later case, compares x with previous three. How many comparisons are done in the worst case by this algorithm?
2. How can you modify binary search to eliminate unnecessary work if you are certain that x is in the list? Find the worst case and average case time complexity of this modified algorithm. (you may assume that $n=2^k-1$ for some k)



Exercises



3*. The first n cells of an array L contain integers sorted in increasing order. The remaining cells all contain some very large integer that we may think of as infinity. The array may be arbitrarily large (you may think of it as infinite) and you don't know n . Give an algorithm to find the position of a given integer x ($x < \text{maxint}$) in the array in $\log n$ time.

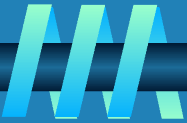
4. What is the largest number of key comparisons made by binary search for a key in the following array?

3 14 27 31 39 42 55 70 74 81 85 93 98

List all the elements in the array that will require largest number of key comparisons.



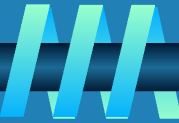
Exercises



5. Sequential search can be used with about same efficiency whether the list is an array or linked list. Is it also true for binary search? If not, explain the time complexity of binary searching a linked list of sorted elements.
6. How can one use binary search for range searching, for finding all the elements in a sorted array whose values fall between two given values L and U (inclusively) $L \leq U$. What is the worst case efficiency of the algorithm?
7. Write a code to find maximum in a 2-3 tree. State the time complexity of the algorithm.



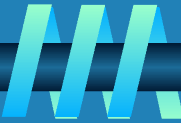
Exercises



8. Write a code to find successor if a node is a 2-3 tree. 3*.9.
What is the time complexity of your algorithm.
10. How would you check if the given tree is a 2-3 tree. What is the time complexity of your algorithm.
11. Why are balanced trees useful?
12. Use a good hashing functions to store all the words in the first four lines of the following poem. Is your search time 1?
(do not use probing technique)



Exercises



12. Use a good hashing functions to store all the words in the first four lines of the following poem. Is your search time 1? (do not use probing technique)

“ Twinkle Twinkle little star
How I wonder what you are
Up above the world so high
Like a diamond in the sky”

13. Insert the words in the poem above using both linear and quadratic probing. You may use string length as the data you will insert. Then search for “high” and “star”, “sky” and “you”. How many probes were needed in linear probing? How many were needed in quadratic probing?

