CS213/293 Data Structure and Algorithms 2024

Lecture 1: Why should you study data structures?

Instructor: Ashutosh Gupta

IITB India

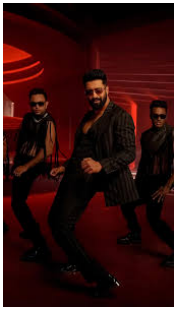
Compile date: 2024-08-29

Next course in programming

Are CS101 and SSL not enough to be a programmer?

In CS101, you learned to walk.

In this course, you will learn to dance.



What is data?

Things are not data, but information about them is data.

Example 1.1

Age of people, height of trees, price of stocks, and number of likes.

Data is big!

We are living in the age of big data!

*Image is from the Internet.

Exercise 1.1

1. *Estimate the number of messages exchanged for status level in Whatsapp.*
2. *How much text data was used to train ChatGPT?*

We need to work on data

We process data to solve our problems.

Example 1.2

1. *Predict the weather*
2. *Find a webpage*
3. *Recognize fingerprint*

Disorganized data will need a lot of time to process.

Exercise 1.2

How much time do we need to find an element in an array?

Problems

Definition 1.1

A problem is a pair of an input specification and an output specification.

Example 1.3

The problem of search consists of the following specifications

- *Input specification: an array S of elements and an element e*
- *Output specification: position of e in S if it exists. If it is not found, return -1.*

Output specifications refer to the variables in the input specifications

Exercise 1.3

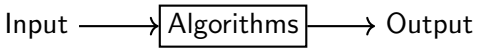
According to the specification, what should happen if e occurs multiple times in S?

Algorithms

Definition 1.2

An algorithm solves a given problem.

- Input $\in$ Input specifications
- Output $\in$ Output specifications



Note: There can be many algorithms to solve a problem.

Exercise 1.4

1. *What is an algorithm?*
2. *How is it different from a program?*

Commentary: An algorithm is a step-by-step process that processes a small amount of data in each step and eventually computes the output. The formal definition of the algorithm will be presented to you in CS310. It took the genius of Alan Turing to give the precise definition of an algorithm.

Example: an algorithm for search

Example 1.4

```
int search( int* S, int n, int e) {
  // n is the length of the array S
  // We are looking for element e in S
  for( int i=0; i < n; i++ ) {
    if( S[i] == e ) {
      return i;
    }
  }
  return -1; // Not found
}
```

Exercise 1.5

What is the run time of the above algorithm if e is not in S?

Commentary: Answer: We count memory accesses, arithmetic operations (including comparisons), assignments, and jumps. The loop in the program will iterate n times. In each iteration, there will be one memory access `S[i]`, three arithmetic operations `i<n`, `S[i] == e` and `i++`, and two jumps. At the initialization, there is an assignment `i=0`. For the loop exit, there will be one more comparison and jump. $Time = nT_{Read} + (3n+2)T_{Arith} + (2n+1)T_{jump} + T_{return}$ Give this program to https://godbolt.org/ and see the assembly. Check if the above analysis is faithful!

Data needs structure

Storing data as a pile of stuff, will not work. We need structure.

Example 1.5

Store files in the order of the year. How do we store data at IIT Bombay Hospital?

Structured data helps us solve problems faster

We can exploit the structure to design efficient algorithms to solve our problems.

The goal of this course!

Example: search on well-structured data

Example 1.6

Let us consider the problem of search consisting of the following specifications

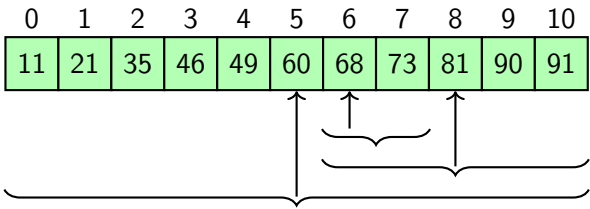
- *Input specification: a non-decreasing array S and an element e*
- *Output specification: Position of e in S. If not found, return -1.*

Example: search on well-structured data

Let us see how we can exploit the structured data!

Let us try to search 68 in the following array.

- Look at the middle point of the array.
- Since the value at the middle point is less than 68, we search only in the upper half.
- We have halved our search space.
- We recursively half the space.



A better search

Example 1.7

```
int BinarySearch(int* S, int n, int e){
  // S is a sorted array
  int first = 0, last = n;
  int mid = (first + last) / 2;
  while (first < last) {
    if (S[mid] == e) return mid;
    if (S[mid] > e) {
      last = mid;
    } else {
      first = mid + 1;
    }
    mid = (first + last) / 2;
  }
  return -1;
```

Commentary: Answer: There will be k iterations. In each iteration, the function will follow the same path. In each iteration, there will be

- a memory access `S[mid]`,(why only one)
- five arithmetic operations `first < last`, `S[mid] == e`, `S[i] > e`, `first+last`, and `../2`,
- one assignment `last = mid`, (Why?)
- three jumps because of two ifs and a loop exit,

For loop exit, there will be one additional comparison and a jump at the loop head. In the initialization section, we have two assignments and two arithmetic operations.
$Time = kT_{Read} + (6k+5)T_{Arith} + (3k+1)T_{jump} + T_{return}$

Exercise 1.6

Let $n = 2^{k-1}$. How much time will it take to run the above algorithm if `S[0] > e`?

Topic 1.1

Big-O notation

How much resource does an algorithm need?

There can be many algorithms to solve a problem.

Some are good and some are bad.

Good algorithms are efficient in

- time and
- space.

Our method of measuring time is cumbersome and machine-dependent.

We need approximate counting that is machine-independent.

Commentary: Sometimes there is a trade-off between time and space. For example, the inefficient linear search only needed one extra integer, but the binary search used three extra integers. The difference between two integers may be a minor issue, but it illustrates the trade-off.

Input size

An algorithm may have different running times for different inputs.

How do we think about comparing algorithms?

We define the rough size of the input, usually in terms of important parameters of input.

Example 1.8

In the problem of search, we say that the number of elements in the array is the input size.

Please note that the size of individual elements is not considered. *(Why?)*

Commentary: Ideally, the number of bits in the binary representation of the input is the size, which is too detailed and cumbersome to handle. In the case of search, we assume that elements are drawn from the space of size 2^{32} and can be represented using 32 bits. Therefore, the type of the element was `int`.

Best/Average/Worst case

For a given size of inputs, we may further make the following distinction.

1. Best case: Shortest running time for some input.
2. Worst case: Worst running time for some input.
3. Average case: Average running time on all the inputs of the given size.

Exercise 1.7

How can we modify almost any algorithm to have a good best-case running time?

Example: Best/Average/Worst case

Example 1.9

```
int BinarySearch(int* S, int n, int e){
  // S is a sorted array
  int first = 0, last = n;
  int mid = (first + last) / 2;
  while (first < last) {
    if (S[mid] == e) return mid;
    if (S[mid] > e) {
      last = mid;
    } else {
      first = mid + 1;
    }
    mid = (first + last) / 2;
  }
  return -1;
}
```

In `BinarySearch`, let $n = 2^{k-1}$.

1. Best case: $e == S[n/2]$
   $T_{Read} + 6T_{Arith} + T_{return}$,
2. Worst case:$e \notin S$
   We have seen the worst case.
3. The average case is roughly equal to the worst case because most often the loop will iterate k times. (Why?)

Commentary: Analyzing the average case is usually involved. For some important algorithms, we will do a detailed average time analysis.

Asymptotic behavior

For short inputs, an algorithm may use a shortcut for better running time.

To avoid such false comparisons, we look at the behavior of the algorithms in limit.

Ignore hardware-specific details

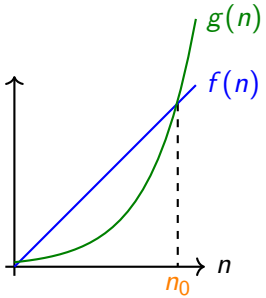
- Round numbers $100000000000001 \approx 100000000000000$
- Ignore coefficients $3kT_{Arith} \approx k$

Big-O notation: approximate measure

Definition 1.3

Let f and g be functions $\mathbb{N} \to \mathbb{N}$. We say $f(n) \in O(g(n))$ if there are c and n_0 such that

$$f(n) \leq cg(n) \qquad \text{for all } n \geq n_0.$$



- In limit, $cg(n)$ will dominate $f(n)$
- We say $f(n)$ is $O(g(n))$

Exercise 1.8

Which of the following are the true statements?

- $5n + 8 \in O(n)$
- $5n + 8 \in O(n^2)$
- $5n^2 + 8 \in O(n)$
- $n^2 + n \in O(n^2)$
- $5000000000000000000000000n^2 \in O(n^2)$
- $50n^2 log n + 60n^2 \in O(n^2 log n)$

Example: Big-O of the worst case of `BinarySearch`

Example 1.10

In `BinarySearch`, let $n = 2^{k-1}$.

1. Worst case: $e \notin S$
   $kT_{Read} + (6k+5)T_{Arith} + (3k+1)T_{jump} + T_{return} \in O(k)$

   Since $k = log\ n + 1$, therefore $k \in O(log\ n)$

   We may also say `BinarySearch` is $O(log\ n)$.

   Therefore, the worst-case running time of `BinarySearch` is $O(log\ n)$.

Exercise 1.9

Prove that $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.

What does Big O says?

Expresses the approximate number of operations executed by the program as a function of input size

Hierarchy of algorithms

- $O(log\ n)$ algorithm is better than $O(n)$
- We say $O(log\ n) < O(n) < O(n^2) < O(2^n)$

May hide large constants!!

Complexity of a problem

The complexity of a problem is the complexity of the best-known algorithm for the problem.

Exercise 1.10

What is the complexity of the following problem?

- *sorting an array* — $O(n^2)$ ✗
- *matrix multiplication* — $O(n^3)$ ✗

Best algorithm is still not known

Exercise 1.11

What is the best-known complexity for the above problems?

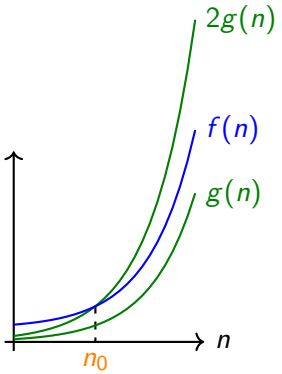
Commentary: A discussion on the latest developments in matrix multiplication algorithms. https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication

Θ-Notation

Definition 1.4 (Tight bound)

Let f and g be functions $\mathbb{N} \to \mathbb{N}$. We say $f(n) \in \Theta(g(n))$ if there are c_1, c_2, and n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \text{for all } n \geq n_0.$$



There are more variations of the above definition. Please look at the end.

Exercise 1.12

a. Does the worst-case complexity of `BinarySearch` belong to $\Theta(\log n)$?

b. If yes, give c_1, c_2, and n_0 for the application of the above definition on `BinarySearch`.

Names of complexity classes

- Constant: $O(1)$
- Logarithmic: $O(logn)$
- Linear: $O(n)$
- Quadratic: $O(n^2)$
- Polynomial : $O(n^k)$ for some given k
- Exponential : $O(2^n)$

Topic 1.2

Tutorial Problems

Problem: Compute the exact running time of insertion sort.

Exercise 1.13

The following is the code for insertion sort. Compute the exact worst-case running time of the code in terms of n and the cost of doing various machine operations.

```
for( int j = 1; j < n; j++ ) {
  int key = A[j];
  int i = j-1;
  while( i >= 0 ) {
    if( A[i] > key ) {
      A[i+1] = A[i];
    }else{
      break;
    }
    i--;
  }
  A[i+1] = key;
}
```

Problem: additions and multiplication

Exercise 1.14

What is the time complexity of binary addition and multiplication? How much time does it take to do unary addition?

Problem: hierarchy of complexity

Exercise 1.15

Given $f(n) = a_0 n^0 + ... + a_d n^d$ *and* $g(n) = b_0 n^0 + ... + b_e n^e$ *with* $d > e$ *and* $a_d > 0$ *(Why?), show that* $f(n) \notin O(g(n))$.

Topic 1.3

Problems

Order of functions

Exercise 1.16

- *If* $f(n) \leq F(n)$ *and* $G(n) \geq g(n)$ *(in order sense) then show that* $\frac{f(n)}{G(n)} \leq \frac{F(n)}{g(n)}$.
- *Is* $f(n)$ *the same order as* $f(n)|sin(n)|$*?*

Exercise: an important complexity class!

Exercise 1.17

Prove that $O(\log(n!)) = O(n \log n)$. *Hint: Stirling's approximation*

Exercise: Egg drop problem (Quiz 2024)

Exercise 1.18

In the dead of night, a master jewel thief is plotting the heist of a lifetime-stealing the most valuable Faberge Egg from a towering 100-story museum. Each floor of the building has an identical egg, but the higher the floor, the more valuable the egg becomes. However, there's a catch. The thief can steal only one egg and she knows that the most valuable egg at the top may not survive a drop from such a great height. To avoid smashing her prized loot, she must identify the highest floor from which an egg can be dropped without breaking. Armed with two replica eggs from the museum's gift shop-perfectly identical but utterly worthless-the thief devises a plan. These two eggs will be her test subjects, sacrificed in the pursuit of the perfect drop. But time is of the essence, and the thief can not afford to be caught by the museum guards. She needs to figure out the minimum number of test drops required to guarantee finding the highest safe floor. Once an egg is broken, it's gone for good-no replacements, no second chances. She cannot use any other method to determine the sturdiness of the eggs.

Give an algorithm for the thief to determine, with the least number of drops in the worst case, the highest floor from which an egg can be safely dropped without breaking?

Commentary: https://www.youtube.com/watch?v=NGtt7GJ1uiM

Topic 1.4

Extra slides: More on complexity

Ω notation

Definition 1.5 (Lower bound)

Let f and g be functions $\mathbb{N} \to \mathbb{N}$. We say $f(n) \in \Omega(g(n))$ if there are c and n_0 such that

$$cg(n) \leq f(n) \qquad \text{for all } n \geq n_0.$$

Small-o,ω notation

Definition 1.6 (Strict Upper bound)

Let f and g be functions $\mathbb{N} \to \mathbb{N}$. We say $f(n) \in o(g(n))$ if for each c, there is n_0 such that

$$f(n) \leq cg(n) \qquad \text{for all } n \geq n_0.$$

Definition 1.7 (Strict Lower bound)

Let f and g be functions $\mathbb{N} \to \mathbb{N}$. We say $f(n) \in \omega(g(n))$ if for each c, there is n_0 such that

$$cg(n) \leq f(n) \qquad \text{for all } n \geq n_0.$$

Exercise 1.19

a. Prove that $f \in o(g)$ implies $f \in O(g)$.
b. Show that $f \in O(g)$ does not imply $f \in o(g)$.

Size of functions

We can define a partial order over functions using the above notations

- $f(n) \in O(g(n))$ implies $f(n) \leq g(n)$
- $f(n) \in o(g(n))$ implies $f(n) < g(n)$
- $f(n) \in \Omega(g(n))$ implies $f(n) \geq g(n)$
- $f(n) \in \omega(g(n))$ implies $f(n) > g(n)$
- $f(n) \in \Theta(g(n))$ implies $f(n) = g(n)$

Exercise 1.20

Show that the partial order is well-defined.

Commentary: Why do we need to prove that the definition is well-defined?

End of Lecture 1

CS213/293 Data Structure and Algorithms 2024

Lecture 2: Containers in C++

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-08-04

What are containers?

A collection of C++ objects

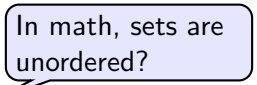
- `int a[10]; //Array`
- `vector<int> b;`

Exercise 2.1

Why the use of the word 'containers'?

More container examples

- `array`
- `vector<T>`
- `set<T>`
- `map<T,T>`
- `unordered_set<T>`
- `unordered_map<T,T>`



Set in C++ $\not\equiv$ Mathematical set

Why do we need containers?

Collections are everywhere

- CPUs in a machine
- Incoming service requests
- Food items on a menu
- Shopping cart on a shopping website

Not all collections are the same

Example: using a container

Source: http://www.cplusplus.com/reference/set

```
#include <iostream>
#include <set>
int main () {
  std::set<int> s;
  for(int i=5; i>=1; i--)    // s: {50,40,30,20,10}
    s.insert(i*10);
  s.insert(20);     // no new element inserted
  s.erase(20);     // s: {50,40,30,10}

  if( s.contains(40) )
    std::cout << "s has 40!\n";

  for( int i : s )  // printing elements of a container
    std::cout << i << '\n';
  return 0;
}
```

Why do we need many kinds of containers?

- Expected properties and usage patterns define the container

  For example,
  - Unique elements in the collection
  - Arrival/pre-defined order among elements
  - Random access vs. sequential access
  - Only few additions(small collection) and many membership checks
  - Many additions (large collection) and a few sporadic deletes

Different containers are

efficient to use/run

in varied usage patterns

Choose a container

Exercise 2.2

Which container should we use for the following collections?

- *CPUs in a machine*
- *Incoming service requests*
- *Food items on a menu*
- *Shopping cart on a shopping website*

Some examples of containers

`set<T>`

- Unique element
- insert/erase/contains interface
- collection has implicit ordering among elements

`map<T,T>`

- Unique key-value pairs
- insert/erase interface
- collection has implicit ordering among keys
- Finding a key-value pair is not the same as accessing it
- Throws an exception if accessed using a non-existent key

Containers are abstract data types

The containers do not provide details on the implementation. They provide an interface with guarantees.

In computer science, we call the libraries abstract data types. The guarantees are called axioms of abstract data type.

Commentary: Defining the axioms is not a simple matter. We need to answer the following questions.

Why do we need exactly these five axioms?
Are these sufficient?
Are any of them redundant, i.e., implied by others?
Do they contradict each other?

These kind of questions will be answered in CS228.

Example 2.1

Axioms of abstract data type set.

- `std::set<int> s; s.contains(v) == false`
- `s.insert(v); s.contains(v) == true`
- `x = s.contains(u); s.insert(v); s.contains(u) == x`, where $u! = v$.
- `s.erase(v); s.contains(v) == false`
- `x = s.contains(u); s.erase(v); s.contains(u) == x`, where $u! = v$.

Example: map<T,T>

Source: http://www.cplusplus.com/reference/map

```
#include <iostream>
#include <string>
#include <map>
int main () {
  std::map<std::string,int> cart;
  //Set some initial values:
  cart["soap"] = 2;
  cart["salt"] = 1;
  cart.insert( std::make_pair( "pen", 10 ) );
  cart.erase("salt");
  //access elements
  std::cout << "Soap: " << cart["soap"] << "\n";
  std::cout << "Hat: " << cart["hat"] << "\n";
  std::cout << "Hat: " << cart.at("hat") << "\n";
}
```

Commentary: When we run `cart["hat"]`, C++ modifies the content of cart and maps "hat" to 0 (default value of int). Therefore, the run `cart.at("hat")` succeeds without exception. If we delete the second last statement containing `cart["hat"]` in the program, the last statement will throw an exception. It is a strange situation, where mere reading a data structure is modifying it and changing the behavior of the data structure.

Exercise 2.3 *What will happen at the last two calls?*

Exceptions in Containers

If containers cannot return an appropriate value, they throw exceptions.

Callers must be ready to catch the exceptions and respond accordingly.

Example 2.2

Read operation `cart.at("shoe")` throws an exception if the cart does not value for key `"shoe"` .

STL: container libraries with unified interfaces

Since the containers are similar

`http://www.cplusplus.com/reference`

C++ in flux

Once C++ was set in stone. Now, modern languages have made a dent!

Three major revisions in history!!

- c++98
- c++11
- c++17
- c++20 (we will use this compiler!)

Daily Quiz

```
#include <iostream>
#include <map>
int main() {
  std::map<int, std::string> responses = {  {0, "Zero value!"},
                    {-1, "Negative one!"}, {2, "Positive two!"} };
  int x;
  std::cin >> x;
  if (responses.find(x) != responses.end()) {
    std::cout << responses[x] << std::endl;
  } else {
    std::cout << "Default response." << std::endl;
  }
  return 0;
}
```

Topic 2.1

Exceptions

What to do if an unexpected event occurs

Example 2.3

- Divide by zero
- Open a non-existent file
- Network device is failed

Stop the program and throw an exception!

Exceptions: something unexpected happened!

```
#include <iostream>
using namespace std;

int foo(int x) {
  try
  {
    throw 20; // something has gone wrong!!
  }
  catch (int e) // type of e must match the type of thrown value!
  {
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
```

Exceptions: catch matches the types!

```
int foo(int x) {
  try{
    if( x >  0 ){
      throw 20; // something has gone wrong!!
    }else{
      throw "C'est la vie!"; // Another thing has gone wrong!
    }
  }
  catch (int e){ // type of e is matched!
    cout << "An int exception occurred. " << e << '\n';
  }
  catch (string e){ // type of e is matched!
    cout << "A string exception occurred. " << e << '\n';
  }
  return 0;
}
```

Exceptions in the callee

```
int bar(){
  ...
  throw 20; // something has gone wrong!!
  ...
}

int foo(int x) {
  try{
    bar();
  }
  catch (int e){ // type of e is matched!
    cout << "An int exception occurred. " << e << '\n';
  }
}
```

Why write exceptions instead of handling the "unexpected" cases?

To avoid cumbersome code!

If no catch is written, the exception flows to the top, and the program fails.

Exceptions provide a succinct mechanism to handle all possible errors, with a few catches.

Topic 2.2

Array vs. Vector

Vector

- Variable length
- Primarily stack-like access
- Allows random access
- Difficult to search
- Overhead of memory management

Array

- Fixed length
- Random access
- Difficult to search
- Low overhead

Let us create a test to compare the performances

```
#include <iostream>
#include <vector>
#include "rdtsc.h"
using namespace std; // unclear!! STOP ME!
int local_vector(size_t N) {
  vector<int> bigarray; //initially empty vector
  //Fill vector up to length N
  for(unsigned int k = 0; k<N; ++k)
    bigarray.push_back(k);
  //Find the max value in the vector
  int max = 0;
  for(unsigned int k = 0; k<N; ++k) {
    if( bigarray[k] > max )
      max = bigarray[k];
  }
  return max;
} // 3N memory operations
```

Let us create a test to compare the performance (2)

```
// call local_vector M times
int test_local_vector( size_t M, size_t N ) {
  unsigned sum = 0;
  for(unsigned int j = 0; j < M; ++j ) {
    sum = sum + local_vector( N );
  }
  return sum;
}
//In total, 3MN memory operations
```

Let us create a test to compare the performance (3)

```
// assumes the 64-bit machine
int main() {
  ClockCounter t; // counts elapsed cycles
  size_t MN = 4*32*32*32*32*16;
  size_t N = 4;
  while( N <= MN  ) {
    t.start();
    test_local_vector( MN/N , N );
    double diff = t.stop();
    //print average time for 3 memory operations
    std::cout << "N = " << N << " : "<< (diff/MN);
    N = N*32;
  }
}
```

Exercise 2.4

Write the same test for arrays.

Topic 2.3

Tutorial Problems

Exercise: What is the difference between at and ..[..] accesses?

Exercise 2.5

What is the difference between "at" and "..[..]" accesses in C++ maps?

Exercise: smart pointers

Exercise 2.6

C++ does not provide active memory management. However, smart pointers in C++ allow us the capability of a garbage collector. The smart pointer classes in C++ are

- `shared_ptr`
- `weak_ptr`
- `unique_ptr`
- `auto_ptr`

Write programs that illustrate the differences among the above smart pointers.

Exercise: const

Exercise 2.7

Why do the following three writes cause compilation errors in the C++20 compiler?

```
class Node {
public:
  Node() : value(0) { }
  const Node& foo( const Node* x) const {
    value = 3;          // Not allowed because of _____________
    x[0].value = 4;     // Not allowed because of _____________
    return x[0];
  }
  int value;
};
int main() {
  Node x[3], y;
  auto& z = y.foo(x);
  z.value = 5; // Not allowed because of _____________
}
```

Topic 2.4

Problems

Exercise: named requirements

Exercise 2.8

Some of the containers have named requirements in their description. For example, "std::vector (for T other than bool) meets the requirements of Container, AllocatorAwareContainer (since C++11), SequenceContainer, ContiguousContainer (since C++17), and ReversibleContainer.".

What are these? Can you describe the meaning of these? How are these conditions checked?

Exercise: auto in exception (2024 student suggestion!)

Exercise 2.9

Can we write auto within the catch parameter?

```
int foo(int x) {
  try{
    throw 20; // something has gone wrong!!
  }
  catch (auto e){ // type of e is matched!
    cout << "An int exception occurred. " << e << '\n';
  }
  return 0;
}
```

Topic 2.5

Extra slides: weak pointers

An illustrative example of weak pointer usage (continued)

```
#include <iostream>
#include <memory>
class Node {
public:
  Node(int value) : value(value) {std::cout << "Node " << value << " created." << std::endl; }
  // Functions to set/get the next node/weak ref to previous node/shared ref to previous node
  void setNext    ( std::shared_ptr<Node> next ) { nextNode = next;     }
  void setWeakPrev( std::shared_ptr<Node> next ) { prevWeakNode = next; }
  void setPrev    ( std::shared_ptr<Node> next ) { prevNode = next;     }
  std::shared_ptr<Node> getNext()     const      { return nextNode;            }
  std::shared_ptr<Node> getPrev()     const      { return prevNode;            }
  std::shared_ptr<Node> getWeakPrev() const      { return prevWeakNode.lock(); }
  // Function to display the value of the node
  void display() const { std::cout << "Node value: " << value << std::endl; }
private:
  int value;
  std::shared_ptr<Node> nextNode;
  std::shared_ptr<Node> prevNode;
  std::weak_ptr<Node>   prevWeakNode;
};
void print_list( std::weak_ptr<Node> current ) {
  for (int i = 0; i < 5; ++i) {
    auto current_ref = current.lock();
    if (current_ref) {
      current_ref->display();
      current = current_ref->getNext();
    } else {
      std::cout << "Next node is nullptr." << std::endl; break;
    }
  }
}
```

An example of weak pointer usage (2)

```
// Creating a doubly linked list via shared_ptr/weak_ptr
std::weak_ptr<Node> shared_test() {
  auto node1 = std::make_shared<Node>(1);
  auto node2 = std::make_shared<Node>(2);
  auto node3 = std::make_shared<Node>(3);
  // Create a circular reference
  node1->setNext(node2);
  node2->setNext(node3);
  node2->setPrev(node1); // shared pointer pointing to previous node is causing a reference cycle
  node3->setPrev(node2);
  return node1;
}
std::weak_ptr<Node> weak_test() {
  auto node1 = std::make_shared<Node>(1);
  auto node2 = std::make_shared<Node>(2);
  auto node3 = std::make_shared<Node>(3);
  node1->setNext(node2);
  node2->setNext(node3);
  node2->setWeakPrev(node1); // weak pointer pointing to previous node breaks cyclic reference counting
  node3->setWeakPrev(node2);
  return node1;
}
int main() {
  std::cout << "Testing shared pointer:" << std::endl;
  auto current = shared_test();
  print_list(current);
  std::cout << "Testing weak pointer:" << std::endl;
  current = weak_test();
  print_list(current);
  return 0;
}
```

End of Lecture 2

CS213/293 Data Structure and Algorithms 2024

Lecture 3: Stack and queue

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-08-10

Topic 3.1

Stack

Stack

Definition 3.1

Stack is a container where elements are added and deleted according to the last-in-first-out (LIFO) order.

- Addition is called pushing
- Deleting is called popping

Example 3.1

- *Stack of papers in a copier*
- *Undo-redo features in editors*
- *Back button on Browser*

Interface of stack

Reference: https://en.cppreference.com/w/cpp/container/stack

Stack supports four interface methods

- `stack<T> s` : allocates new stack s
- `s.push(e)` : Pushes the given element e to the top of the stack.
- `s.pop()` : Removes the top element from the stack.
- `s.top()` : accesses the top element of the stack.

Some support functions

- `s.empty()` : checks whether the stack is empty
- `s.size()` : returns the number of elements

Exercise 3.1

Why define stack when we can use vector for the same effect?

Commentary: Answer: vector in C++ promises to provide efficient random access but stack does not make such a promise. Therefore, the implementations of stack may make implementation choices that may result in inefficient random access.

Axioms of stack

Let `s1` and `s` be stacks.

- `Assume(s1 == s); s.push(e); s.pop(); Assert(s1==s);`
- `s.push(e); Assert(s.top()==e);`

`Assume(s1 == s)` means that we assume that the content of s1 and s are the same.
`Assert(s1 == s)` means that we check that the content of s1 and s are the same.

Exercise: action on the empty stack

Exercise 3.2

Let `s` be an empty stack in C++.

- What happens when we run `s.top()`?
- What happens when we run `s.pop()`?

Ask ChatGPT.

Commentary: Answer: `s.top()` will cause a segmentation fault. `s.pop()` will not cause any error and exit without any effect.

Topic 3.2

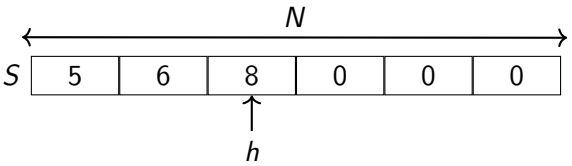
Implementing stack

Array-based stack

Let us look at a simplified array-based implementation of an array of integers.

The stack consists of three variables.

- N specifies the currently available space in the stack
- S is the integer array of size N
- h is the position of the head of the stack



Implementing stack

```
class arrayStack {
  int  N = 2;     // Capacity
  int* S = NULL; // pointer to array
  int  h = -1;    // Current head of the stack
public:
  arrayStack() { S = (int*)malloc(sizeof(int)*N ); }
  int  size()  { return h+1;  }
  bool empty() { return h<0;  }
  int  top()   { return S[h]; } // On empty stack what happens?
  void push(int e) {
    if( size() == N ) expand(); // Expand capacity of the stack
    S[++h] = e;
  }
  void pop() { if( !empty() ) h--; }
```

Commentary: The behavior of the above implementation may not match the behavior of the C++ stack library. To ensure segmentation fault in top() when the stack is empty one may use the following code. `if( empty() ) return *(int*)0; else return S[t];`

Implementing stack (expanding when full)

```
private:
  void expand() {
    int new_size = N*2; // We observed the growth in our lab!!
    int* tmp = (int*) malloc( sizeof(int)*new_size );//New array
    for( unsigned i =0; i < N; i++ ) {// copy from the old array
      tmp[i] = S[i];
    }
    free(S);       // Release old memory
    S = tmp;       // Update local fields
    N = new_size; //
  }
};
```

Efficiency

All operations are performed in $O(1)$ if there is no expansion to stack capacity.

What is the cost of expansion?

Topic 3.3

Why exponential growth strategy?

Growth strategy

Let us consider two possible choices for growth.

- Constant growth: `new_size = N + c` (for some fixed constant c)
- Exponential growth: `new_size = 2*N`

Which of the above two is better?

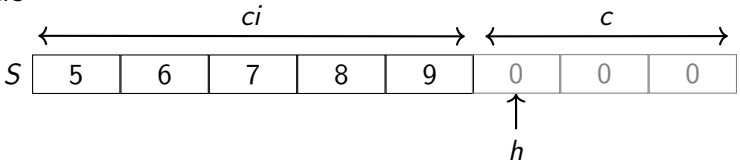
Analysis of constant growth

Let us suppose initially $N = 0$ and there are n consecutive pushes.

After every cth push, there will be an expansion operation.

Therefore, the expansion operation at $(ci + 1)$th push will

- allocate memory of size $c(i + 1)$
- copy ci integers



Cost of ith expansion: $c(2i + 1)$.

Commentary: We are assuming that allocating memory of size k costs k time, which may be more efficient in practice. Bulk memory copy can also be sped up by vector instructions.

Analysis of constant growth(2)

For n pushes, there will be n/c expansions.

The total cost of expansions:

$$c(1 + 3 + ... + (2\frac{n}{c} + 1)) = c(n/c)^2 \in O(n^2)$$

Non-linear cost!

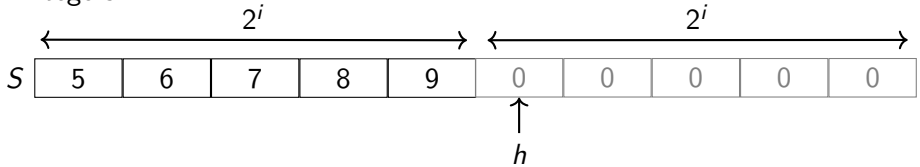
Analysis of exponential growth

Let us suppose initially $N = 1$ and there are $n = 2^r$ consecutive pushes.

The expansion operations will only occur at $2^i + 1$th push, where $i \in [0, r-1]$.

The expansion operation at $2^i + 1$th push will

- allocate memory of size 2^{i+1}
- copy 2^i integers



Cost of the expansion: $3 * 2^i$.

Analysis of exponential growth(2)

For 2^r pushes, the last expansion would be at $2^{r-1} + 1$.

The total cost of expansions:

$$3(2^0 + ... + 2^{r-1}) = 3 * (2^r - 1) = 3 * (n - 1)$$

Linear cost! The average cost of push remains $O(1)$.

Exercise 3.3

Why double? Why not triple? Why not 1.5 times? Is there a trade-off?

Topic 3.4

Applications of stack

Stacks are everywhere

Stack is a foundational data structure.

It shows up in a vast range of algorithms.

Example: matching parentheses

Problem:
Given an input text check if it has matching parentheses.

Examples:

- "{a[sic]tik}"✓
- "{a[sic}tik}"✗

```
bool parenMatch(string text ) {
  std::stack<char> s;
  for(char c : text ) {
    if( c == '{' or c == '[' ) s.push(c);
    if( c == '}' or c == ']' ) {
      if( s.empty()       ) return false;
      if( c-s.top() != 2 ) return false;
      s.pop();
    }
  }
  if( s.empty() ) return true;
  return false;
}
```

Topic 3.5

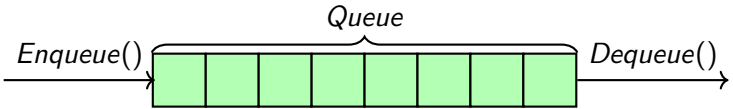
Queue

Queue

Definition 3.2

Queue is a container where elements are added and deleted according to the first-in-first-out (FIFO) order.

- Addition is called enqueue
- Deleting is called dequeue



Example 3.2

- *Entry into an airport*
- *Calling lift in a building (priority queue)*

Interface of queue

Reference: https://en.cppreference.com/w/cpp/container/queue

Queue supports four main interface methods

- `queue<T> q` : allocates new queue q
- `q.enqueue(e)` : Adds the given element e to the end of the queue. (push)
- `q.dequeue()` : Removes the first element from the queue. (pop)
- `q.front()` : access the first element .

Some support functions

- `q.empty()` : checks whether the queue is empty
- `q.size()` : returns the number of elements

Commentary: All literature uses the terms enqueue and dequeue, but unfortunately C++ library uses push for enqueue and pop uses for dequeue. Other languages such as Java uses the terms enqueue and dequeue.

Axioms of queue

1. `queue<T> q; Assert(q.empty() == true);`

2. `q.enqueue(e); Assert(q.empty() == false);`

3. `Assume(q.empty() == true);`
   `q.enqueue(e); Assert(q.front() == e);`

4. `Assume(q.empty() == false && old_q == q);`
   `q.enqueue(e); Assert(old_q.front() == q.front());`

5. `Assume(q.empty() == true && old_q == q);`
   `q.enqueue(e); q.dequeue(); Assert(old_q == q);`

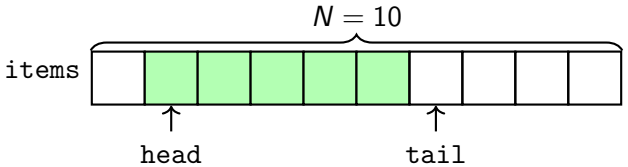
6. `Assume(q.empty() == false && q == q1);`
   `q.enqueue(e);q.dequeue(); q1.dequeue();q1.enqueue(e); Assert(q == q1);`

Topic 3.6

Array implementation of queue

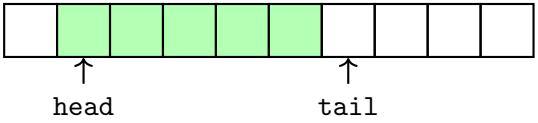
Array-based implementation

- Queue is stored in an array `items` in a circular fashion
- Three integers record the state of the queue
  1. `N` indicates the available capacity (N-1) of the queue
  2. `head` indicates the position of the front of the queue
  3. `tail` indicates position one after the rear of the queue

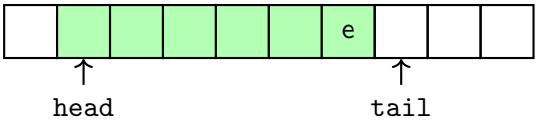


Enqueue operation on array

Consider the state of the queue

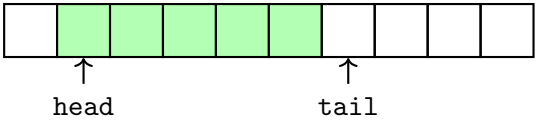


After the enqueue(e) operation:

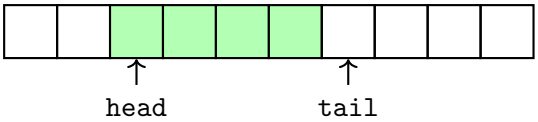


Dequeue operation on array

Consider the state of the queue



After dequeue() operation:

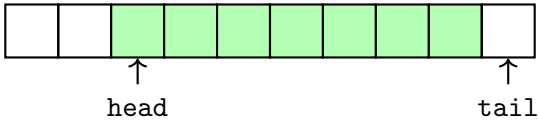


Exercise 3.4

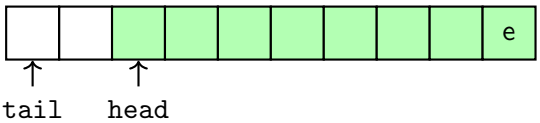
1. *Where will front() read from?*
2. *What is the size of the queue?*

Wrap around to utilize most of the array

Consider the state of the queue

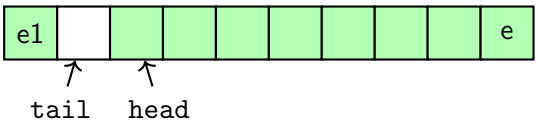


After enqueue(e) operation, we move the tail to 0.



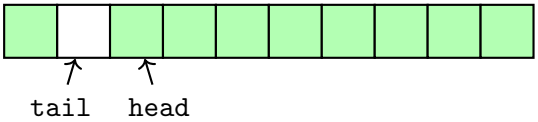
Wrap-around allows us to use the array repeatedly.

After another `enqueue(e1)` operation:

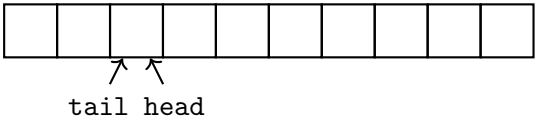


Full and empty queue

Full queue:



Empty queue:



Exercise 3.5

Can we use all N cells for storing elements?

Array implementation

The code is not written in C++; We will slowly move towards pseudo code to avoid clutter on slides.

```
int head = 0, tail=0, N = INITIAL_CAPCITY;

Object items[N];                    //Some initial size

bool empty()   { return (head == tail);   }

bool size()    { return (N+tail-head)%N; }

Object front() { return items[head];      }
```

Array implementation

```
void dequeue() {
  if( empty() ) throw Empty;                  // Queue is empty
  free(items[head]);items[head] = NULL; // Clear memory
  head = (head+1)%N;       //Remove an element
}

void enqueue( Object x ) {
  if ( size() == N-1 ) expand(); // Queue is full;expand
  items[tail] = x;
  tail = (tail+1)%N; // insert element
}
```

Exercise 3.6

In our stack implementation, we did not invoke free in pop, but we invoke free in dequeue. Why?

Commentary: In the stack implementation, we were only handling stack of int. Here, we are handling stack of arbitrary objects. If the object has dynamic size then it cannot live on the stack itself. We will store a reference to the object on stack and the object will be allocated somewhere else. Therefore. the objects must be freed on dequeue (the above syntax of free is not in C++; It will only work when items[head] is a reference). However, there is no need to free an int because it has fixed size and it was stored on the array of the stack itself.

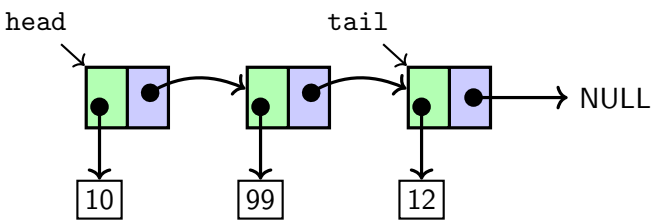
Topic 3.7

Queue via linked list

Linked lists

Definition 3.3

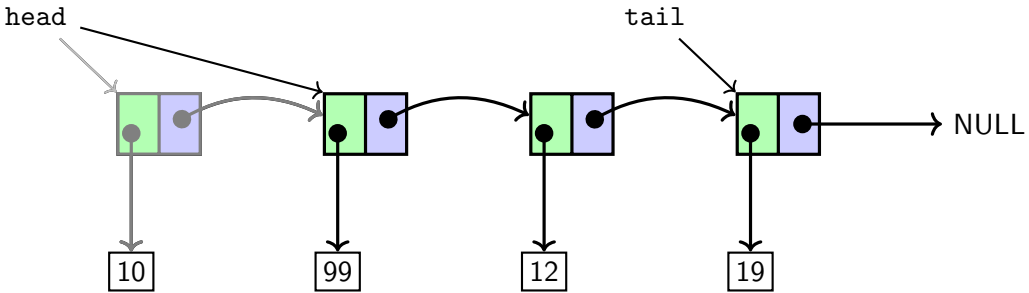
A linked list consists of nodes with two fields data and next pointer. The nodes form a chain via the next pointer. The data pointers point to the objects that are stored on the linked list.



Exercise 3.7

If we use a linked list for implementing a queue, which side should be the front of the queue?

Dequeue in linked lists

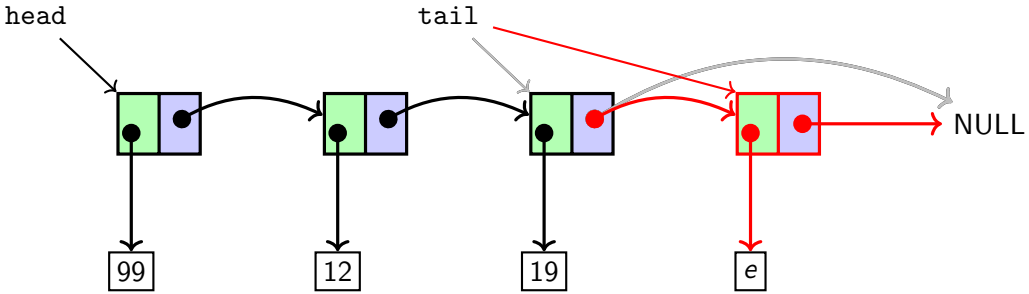


Exercise 3.8

What happens to the object containing 10?

Commentary: Answer: There are several choices. The object is deallocated, the reference to the object is returned to the caller, or the copy of the the object is returned to the caller. Different implementations may do it differently. This behavior is not part of the specification of queue.

Enqueue(e) in linked lists



Exercise 3.9

a. *Which one is better: array or linked list?*

b. *Do we need the tail pointer?*

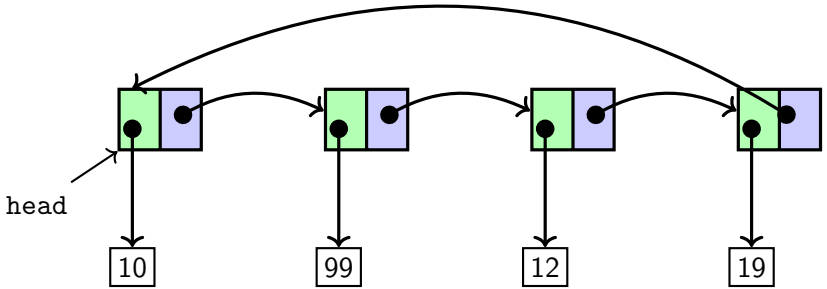
Topic 3.8

Circular linked list

Circular linked lists

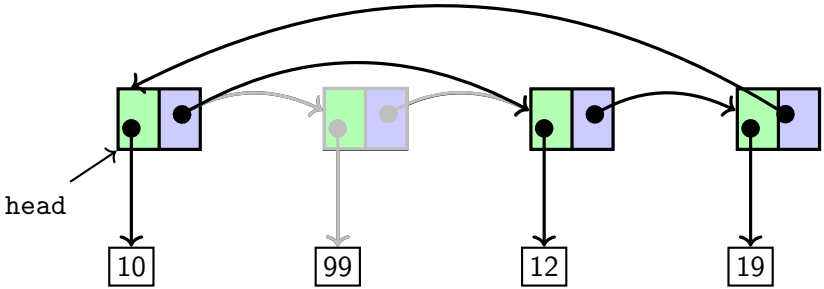
Definition 3.4

In a circular linked list, the nodes form a circular chain via the next pointer.

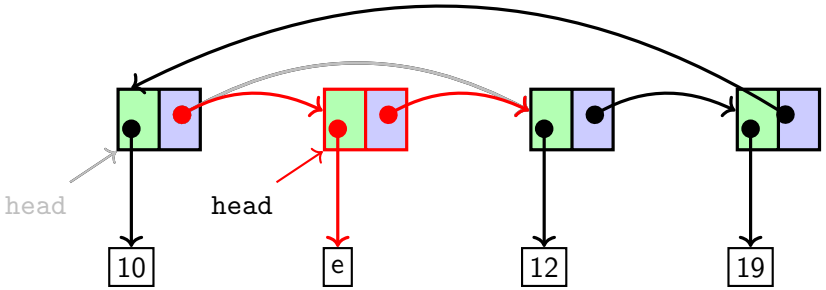


A head pointer points at some node of the circular list. A single pointer can do the job of the head and tail.

Dequeue in circular linked lists



enqueue(e) in circular linked lists



Exercise 3.10

a. *Which element should be returned by front()?*

b. *Give pseudo code of the implementation of queue using circular linked list. (Midsem 2023)*

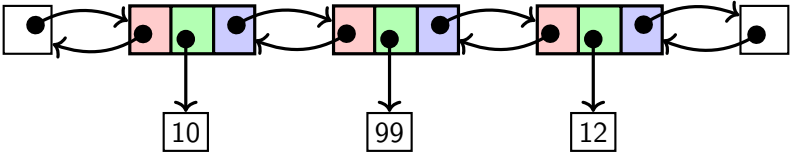
Topic 3.9

Deque via a doubly linked list

Doubly linked lists

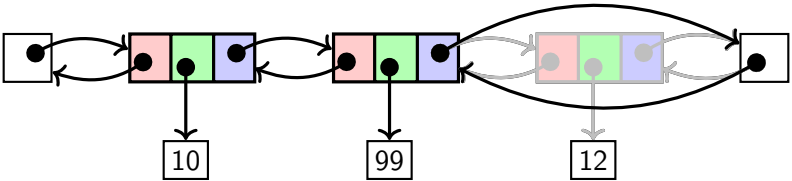
Definition 3.5

A doubly linked list consists of nodes with three fields prev, data, and next pointer. The nodes form a bidirectional chain via the prev and next pointer. The data pointers point to the objects that are stored on the linked list.



At both ends, two dummy or sentinel nodes do not store any data and are used to store the start and end points of the list.

Deleting a node in a doubly linked list



Deque (Double-ended queue)

Definition 3.6

Deque is a container where elements are added and deleted according to both last-in-first-out (LIFO) and first-in-first-out (FIFO) order.

Interface of Deque

Reference: https://en.cppreference.com/w/cpp/container/deque

Queue supports four main interface methods

- `deque<T> q` : allocates new queue q
- `q.push_back(e)` : Adds the given element e to the back.
- `q.push_front(e)` : Adds the given element e to the front.
- `q.pop_front()` : Removes the first element from the queue.
- `q.pop_back()` : Removes the last element from the queue.
- `q.front()` : access the first element .
- `q.back()` : access the first element .

Some support functions

- `q.empty()` : checks whether the stack is empty
- `q.size()` : returns the number of elements

We can implement the Deque data structure using the doubly linked lists.

Stack and queue via Deque

We can implement both stack and queue using the interface of deque.

Exercise 3.11

- *Which functions of deque implement stack?*
- *Which functions of deque implement queue?*

All modification operations are implemented in $O(1)$.

Exercise 3.12

Can we implement `size` in $O(1)$ in a doubly linked list?

Topic 3.10

Tutorial problems

Use of stack

Exercise 3.13

The span of a stock's price on ith day is the maximum number of consecutive days (up to ith day) the price of the stock has been less than or equal to its price on day i.

Example: for the price sequence 2 4 6 3 5 7 *of a stack, the span of prices is* 1 2 3 1 2 6.

Give a linear-time algorithm that computes s_i for a given price series.

Flipping Dosa

Exercise 3.14

There is a stack of dosas on a tava, of distinct radii. We want to serve the dosas of increasing radii. Only two operations are allowed: (i) serve the top dosa, (ii) insert a spatula (flat spoon) in the middle, say after the first k, hold up this partial stack, flip it upside-down, and put it back. Design a data structure to represent the tava, input a given tava, and produce an output in sorted order. What is the time complexity of your algorithm?
This is also related to the train-shunting problem.

Exponential growth

Exercise 3.15

a. Analyze the performance of exponential growth if the growth factor is three instead of two. Does it give us better or worse performance than doubling policy?

b. Can we do a similar analysis for growth factor 1.5?

Problem: reversing a linked list

Exercise 3.16

Give an algorithm to reverse a linked list. You must use only three extra pointers.

Problem: middle element

Exercise 3.17

Give an algorithm to find the middle element of a singly linked list.

Stack and queue (Endsem 2023)

Exercise 3.18

Given two stacks $S1$ and $S2$ (working in the LIFO method) as black boxes, with the regular methods: "Push", "Pop", and "isEmpty", you need to implement a Queue (specifically : Enqueue and Dequeue working in the FIFO method). Assume there are n Enqueue/ Dequeue operations on your queue. The time complexity of a single method Enqueue or Dequeue may be linear in n, however the total time complexity of the n operations should also be $\Theta(n)$.

Topic 3.11

Problems

Problem: messy queue

Exercise 3.19

The mess table queue problem: There is a common mess for k hostels. Each hostel has some N_1,...,N_k students. These students line up to pick up their trays in the common mess. However, the queue is implemented as follows: If a student sees a person from his/her hostel, she/he joins the queue behind this person. This is the "enqueue" operation. The "dequeue" operation is as usual, at the front. Think about how you would implement such a queue. What would be the time complexity of enqueue and dequeue? Do you think the average waiting time in this queue would be higher or lower than a normal queue? Would there be any difference in any statistic? If so, what?

Merge sorted queues (Quiz 2023)

Exercise 3.20

Write a time and space efficient algorithm to merge k sorted-linked list in sorted order, each containing the same no of elements?

Exercise: axioms of queue**

Exercise 3.21

Using axioms of queue show that the assert in the following does not fail.

```
queue<int> q,q1;
q.enqueue(2);
q.enqueue(0);
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(3);

q1.enqueue(7);
q1.enqueue(3);
Assert(q == q1);
```

End of Lecture 3

CS213/293 Data Structure and Algorithms 2024

Lecture 4: Dictionary

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-08-20

Topic 4.1

Problem of dictionary

Storing maps/dictionary

Definition 4.1

A Dictionary stores values so that they can be found efficiently using keys.

Example 4.1

A dictionary may contain bank accounts.

- *Bank account number is the key*
- *The information about your account is the value*
  - *current amount, name, address, etc*
- *To take any action on an account, one needs the key*

Dictionary (Map) container

Reference: https://en.cppreference.com/w/cpp/container/map

In C++ and many languages, dictionaries are called maps.

map supports the following interface.

- `map<Key,T> m` : allocates new map m
- `m.at(e)` : access specified value (throws an exception when value is missing)
- `m[key] = e` : Inserts key-value pair.
- `m.erase(key)` : removes key-value pair.

Some support functions

- `m.empty()` : checks whether the map is empty
- `m.size()` : returns the number of key-value pairs

Order over keys

We have two kinds of maps due to two kinds of keys:

- Ordered: keys are compared using less than, greater than, and equality
  - The default map in C++ assumes keys are ordered.

    Reference: `https://en.cppreference.com/w/cpp/container/map`

- Unordered: keys are compared only using equality
  - For unordered keys, use `unordered_map` in C++.

    Reference: `https://en.cppreference.com/w/cpp/container/unordered_map`

Exercise 4.1

Which data structure are used to store keys in both the maps?

Commentary: Since all data is a bit-vector, C++ can always define order over keys. However, the user should decide if the keys are ordered or unordered. If the keys have no meaning attached to them, we may keep them unordered (while C++ can always default to an ordered map), for example "profile id" on social media. We should use ordered maps, if there is a meaningful order over the keys, "priority of a process" in an OS,

Implementation choices

- arrays, linked lists
- Hash table (`unordered_map` in C++) [Covered in this lecture]
- Binary trees
- Red/black trees (map in C++) [Will be covered in a few lectures!]
- AVL trees
- B-trees

Commentary: AVL trees and B-Trees are not covered in this version of the course. They are the important topics of computer science undergraduate curriculum. Please study them yourself after finishing this course.

Actions on dictionary

We need to design a dictionary data structure keeping in mind the following three important actions on dictionaries.

- Insertion
- Deletion
- Search

Topic 4.2

Design choices for dictionaries

Cost of searching for keys

We have seen in lecture 1 the cost of searching for the position of a key.

Ordered keys

- Binary search is $O(log\ n)$

Unordered keys

- Linear search is $O(n)$

Dictionaries via unordered keys

[2,10,8,19,34,23]

- Searching and deletion is $O(n)$
- Insertion is $O(1)$

Application: Log files, (frequent insertion, but rare searches and deletion)

Dictionaries via ordered keys on arrays

[2,8,10,19,23,34]

- Searching is $O(log\ n)$
- Insertion and deletion is $O(n)$
  - Need to shift keys before insertion/after deletion

Application: Look-up tables (e.g. precomputed values for trigonometric functions), (frequent searches, but rare insertion and deletion)

Exercise 4.2

Can we use a linked list?

One crazy idea: direct addressing!

Consider application: caller ID. We need a map from phone numbers to names.

We have 10-digit-long phone numbers. So let us allocate an array A of size 10^{10}.

Names are stored at the phone number index.

Null	Ashutosh	Null	Null	Divya	Null
9898927391	9898927392	9898927393	9898927394	9898927395	9898927396

- All operations are $O(1)$
- Huge waste of space.

Commentary: Answer: If we allocate a large amount of memory, we may not have $O(1)$ access to the memory. When we say a read from the memory uses $O(1)$ time, there is an implicit assumption that that the total memory is small enough such that the underlying hardware can provide constant time access.

Exercise 4.3

Do we really *have* $O(1)$ *cost for the actions?*

Topic 4.3

Hash table for unordered keys

Can we improve direct addressing?

Can we somehow avoid the waste of space and still benefit from direct addressing?

Let the table size be m and the number of keys be n.

We will design a data structure, where O(1) is the expected time for all operations and the needed storage is $O(m + n)$.

m is roughly equal to n.

Hashing

We choose a function, called the hash function,

$$h : Keys \rightarrow HashValues$$

such that $|HashValues| = m$.

We use $h(key)$ to index the storage array instead of $keys$.

We assume the time to compute $h(key)$ is $\Theta(1)$.

Example: Hashing

Example 4.2

Suppose we want to store caller IDs of phone numbers from your contacts in your phone.

You probably have less than 1000 contacts.

Let us use $h(number) = (number \bmod 1000)$.

We create an array of 1000 entries and store the contact names as follows. Let us suppose Ashutosh's phone number is 9898927392 and Divya's phone number is 9869755395.

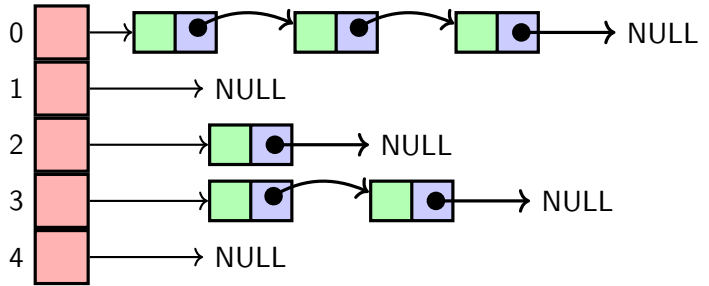
Null	*Ashutosh*	*Null*	*Null*	*Divya*	*Null*
391	*392*	*393*	*394*	*395*	*396*

One problem: Let us suppose Akhil's phone number is 9868733392. We have a collision.

Collision resolution: chaining

In the case of $h(k_1) = h(k_2)$, we cannot store two values in the same place on the array.

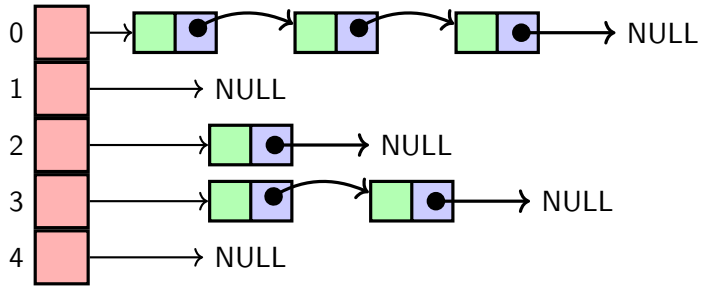
We maintain a linked list for key-value pairs that have the same hash value of their keys and a table (array) indexed by the hash values points to the linked lists.



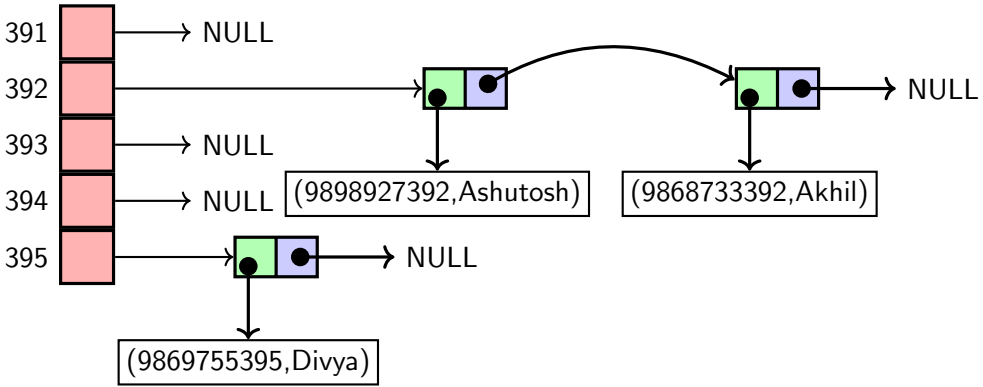
Collision resolution: chaining(2)

To search/insert/delete a (key,value) pair

- using $h(key)$ find position in the table
- search/insert/delete the pair in the linked list of the position.



Example: telephone directory



Topic 4.4

Analysis of hash functions

A good hash function

A good hash function

- distributes keys evenly amongst the positions.
- has a low probability of collision.
- is quick to compute.

Good hash functions are rare - Birthday paradox!

Exercise 4.4

What is a bad hash function?

Load factor

If $n >> m$, there is a greater chance of collisions.

We define load factor $\alpha = \frac{n}{m}$.

Keep α roughly around 1.

- If α is too small, we are wasting space.
- If α is too large, we have long chains.

Exercise 4.5

What to do if α is not known upfront?

Commentary: We start with some default table size. As we utilize more and more of the table, we may resize the table, which may trigger rehashing of the table. Rehashing is an expensive operation.

Simple uniform fictional hash function

- An ideal hash function would pick a position uniformly at random and assign the key to it.
- However, this is not a real hash function, because we will not be able to search later.
- Only for our analysis, we use this simple uniform hash function

Cost of unsuccessful search

- Simple uniform hashing will result in the average list length of α
- Number of elements traversed is α
- Search time is $O(1 + \alpha)$

Cost of successful search

- Assume that a new key-value pair is inserted at the end of the linked list
- Upon insertion of ith key-value pair the expected length of the list is $\frac{i-1}{m}$
- In the case of a successful search of the ith key, the expected number of keys examined is 1 more than the number of keys examined when the ith key-value pair was inserted.
- Expected number of key-value pairs examined for each key search

$$\frac{1}{n}\sum_{i=1}^{n}(1+\frac{i-1}{m}) = 1+\frac{1}{mn}\sum_{i=1}^{n}(i-1) = 1+\frac{1}{mn}\frac{n(n-1)}{2} = 1+\frac{n}{2m}-\frac{1}{2m}$$

- Including the time for computing the hash function we obtain

$$2+\frac{n}{2m}-\frac{1}{2m} \in \Theta(1+\alpha)$$

Topic 4.5

Designing hash functions

Hash function design

$$h : Keys \to \{0, ..., m-1\}$$

m is the size of hash table!

Keys can be of a variety of types.

- Biometric fingerprints
- Addresses
- Words of language dictionaries

Usually, h is the composition of the following functions.

- $encode : Keys \to \mathbb{Z}$
- $compression : \mathbb{Z} \to \{0, ..., m-1\}$

$$h = compression \circ encode$$

Useful functions for encode

- Integer cast: Interpret the bit representation of the key as an integer, if the representation is less than the size of a word (32 bits/64 bits)
- Component sum: If the representation is longer than a word, sum the blocks of 8-bits to compute the integer code.

Example 4.3

$$\textit{encode}(\text{ “Disaster”}) = \text{'D'+ 'i'+'s'+'a'+'s'+'t'+'e'+'r'}$$
$$= 0x44 + 0x69 + 0x73 + 0x61 + 0x73 + 0x74 + 0x65 + 0x72 = 0x33F$$

Example 4.4

Is this a good coding scheme?

Useful functions for encode: polynomial accumulation

- Let $a_0, ..., a_k$ be the list of 8-bit blocks of the binary representation of the *key*.

$$encode(a_0 a_1 .... a_k) = a_0 + a_1 x + a_2 x^2 + ... + a_k x^k$$

  where x is a constant.

- The idea is borrowed from error-correcting codes (e.g. Reed-Solomon codes)

- Observation: the choice of $x = 33, 37, 39$, or 41 gives at most 6 collisions in English vocabulary of 50K+ words. (Please check the claim!)

Exercise 4.6

How can we efficiently compute the polynomial?

Commentary: Usually the polynomial is computed using Horner's rule or precomputed values of x^k. This kind of encoding is widely used in a_i. Overflow is ignored in the computation.

unordered_map in C++ uses Murmurhash2 for encode

```
size_t _Hash_bytes(const char* buf, size_t len, size_t seed) {
  const size_t m = 0x5bd1e995;
  size_t hash = seed ^ len;
  while(len >= 4) { // Mix 4 bytes at a time into the hash.
    size_t k = *((const size_t*)buf);
    k *= m; k ^= k >> 24; k *= m;
    hash *= m; hash ^= k; //something like polynomial accumulation
    buf += 4;len -= 4;
  }
  size_t k;
  switch(len) { // Handle the last few bytes of the input array.
    case 3: k = buf[2]; hash ^= k << 16;
    case 2: k = buf[1]; hash ^= k << 8;
    case 1: k = buf[0]; hash ^= k; hash *= m;
  };
  hash ^= hash >> 13; hash *= m; hash ^= hash >> 15;//Do final mixes.
  return hash;
}
```

Commentary: The above code is from https://github.com/gcc-mirror/gcc/blob/master/libstdc++-v3/libsupc++/hash_bytes.cc

Design of compression

Remainder compression:

$$compression(e) = e \bmod m$$

Here the size of the table matters.

- If $m = 2^k$, the least significant bits of e determine the position in the table. If the output of *encode* is not uniformly distributed, then we do not have enough randomization.
- If m is a prime, $compression(e)$ will return uniformly distributed output. Rule of thumb: stay away from powers of 2.

Example 4.5

Let us suppose, we want to store 2000 *keys and we are ok with three collisions.*

A good choice of m *is* 701, *which is prime near* $2000/3$ *and away from powers of* 2.

Design of compression(2)

Multiplicative compression:

$$\textit{compression}(e) = \lfloor m\{ae\} \rfloor,$$

where $a \in (0, 1)$ is a constant.

- Here the size of the table does not matter.
- However, some values work better than others. Folklore, $\frac{\sqrt{5}-1}{2}$ (golden ratio) works well!

Exercise 4.7
Show $\textit{compression}(e) \in \{0, .., m-1\}$

Commentary: For extended discussion look at The Art of Computer Programming. Volume 3. Sorting and Searching, by Donald Knuth

Design of compression(3)

MAD(multiplication, add, divide) compression:

$$compression(e) = |ak + b| \ mod \ m,$$

where $a, b \in \mathbb{Z}$ are constants.

- Eliminates patterns in input keys if m does not divide a.
- The technique is borrowed from pseudo-random generators!

Topic 4.6

Open addressing: an alternative to chaining!

Open addressing

Open addressing is another way of handling collision.

- The method needs $\alpha \leq 1$
- Each table entry has a key or Null
- We may have to examine many positions for the search

Hash function for open addressing

A slight modification of the hash function.

$$h : Keys \times \{0, .., m-1\} \to \{0, .., m-1\}$$

such that $h(k, 0), ..., h(k, m-1)$ is a permutation of $0, ..., m-1$ for any key k.

Example 4.6

Let $m = 5$.

For some key k,

$$h(k, 0), ..., h(k, 4) = 3, 0, 2, 4, 1.$$

Hash function for open addressing(2)

- $h(key, 0)$ is our usual hash function to place the key.
- $h(key, i)$ is an alternative available choice to place the key if earlier choices $h(key, j)$ for each $j < i$ are occupied.

Open addressing insert

Algorithm 4.1: OpenAddressInsert(k)

```
1 if Table is full then
2     error;
3 i := 0;
4 do
5     probe := h(k,i);
6     i = i + 1;
7 while table[probe] is occupied;
8 table[probe] = k;
```

Linear probing

Linear probing is a special case of open addressing.

In linear probing, we chose h as follows

$$h(k,i) = (h(k,0) + i) \ mod \ m \qquad \text{for each } i > 0.$$

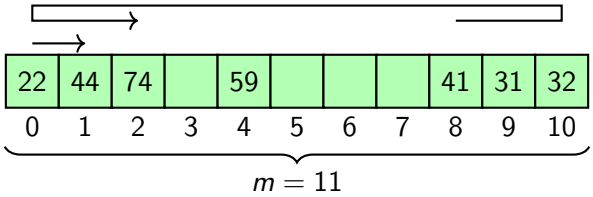
If a position is occupied, take the next one.

Example: insertion in linear probing

Example 4.7

Let $m = 11$ and $h(k, 0) = k \ mod \ 11$.

Let us consider the following sequence of insertions: 41, 22, 44, 59, 32, 31, 74



Open addressing search

Algorithm 4.2: OpenAddressSearch(k)

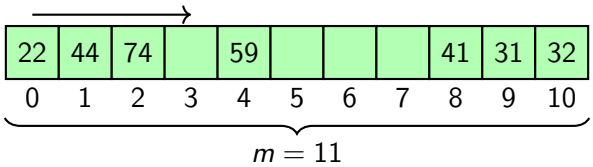
```
1 i := 0;
2 do
3 |   probe := h(k,i);
4 |   if table[probe] == k then
5 |   |   return probe;
6 |   i = i + 1;
7 while (table[probe] is occupied or has tombstone) and i < m;
8 return -1;
```

Example: search in linear probing

Example 4.8

Let $m = 11$ and $h(k, 0) = k \ mod \ 11$.

Let us search for 33 in the following table. We will examine locations from 0 to 3.



Exercise 4.8

How many locations will we examine for the following searches?

- *74*
- *61*
- *44*
- *43*

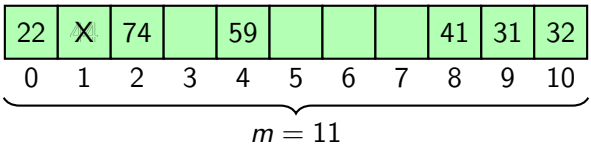
Example: deletion in open addressing

Example 4.9

Let $m = 11$ and $h(k, 0) = k \ mod \ 11$.

Let us delete the key at position 1 in the following table. Will it be correct?

We need to place a marker (tombstone) to indicate that something was here such that we continue to search 74 correctly.



Deletion in open addressing

Algorithm 4.3: OpenAddressDelete(k)

```
1 probe = OpenAddressSearch(k);
2 if probe ≥ 0 then
3     table[probe] = 'X'        // Tombstone marker 'X' indicates that the place was occupied!
```

We can reuse the tombstone location for insertion but assume it is occupied for search.

Exercise 4.9

After many deletions, the performance of the search degrades. How can we recover performance?

Topic 4.7

Tutorial Problems

Problem: probability of collision (Quiz 2023)

Exercise 4.10

What is the probability for the 3rd insertion to have exactly two collisions while using linear probing in the hash table.

Problem: birthday paradox

Exercise 4.11

Given that k elements have to be stored using a hash function with target space n. What is the probability of the hash function having an inherent collision? What is an estimate of the probability of a collision in the insertion of N elements?

Hint: Stirling's approximation $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n+1}} < n! < \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$

Problem: analysis of linear probing

Exercise 4.12

Let $C(i)$ be the chain of array indices that are queried to look for a key k in linear probing where $h(k) = i$.

a. How does this chain extend by an insertion, and how does it change by a deletion?

b. A search for a key k ends when an empty cell is encountered. What if we mark the end of $C(i)$ with an end marker. We stop the search when this marker is encountered. Would this work? Would this be efficient?

c. Is there a way to avoid using tombstones?

Exercise: Double hashing

Exercise 4.13

Let $m = 11$, $h_1(k) = (k \ mod\ 11)$, $h_2 = 6 - (k \mod 6)$.

Let us use the following hash function for an open addressing scheme.

$$h(k, i) = (h_1(k) + i * h_2(k)) \ mod\ m.$$

1. What will be the state of the table after insertions of 41, 22, 44, 59, 32, 31, and 74?
2. Let $h_2(k) = p - (p \mod k)$. What should be the relationship between p and m such that h is a valid function for linear probing?
3. What is the average number of probes for an unsuccessful search if the table has α load factor?
4. What is the average time for a successful search?

Commentary: Double hashing avoids the problem of bunching up the keys, therefore improving search.

Problem: searchable by both keys and values

Exercise 4.14

Suppose you want to store a large set of key-value pairs, for example, (name,address). You have operations, which are addition, deletion, and search of elements in this set. You also have queries whether a particular name or address is there in the set, and if so then count them and delete all such entries. How would you design your hash tables?

Topic 4.8

Extra slides: Binary search in recursive representation!

Search for ordered keys

If keys are stored in order, then we use the binary search that we have discussed in lecture 1.

Algorithm 4.4: BinarySearch(A, key, low, high)

```
1 if low > high then
2     return -1
3 mid := (low+high)/2;
4 if A[mid] == key then
5     return mid
6 if key < A[mid] then
7     return BinarySearch(A,key, low, mid-1)
8 return BinarySearch(A,key, mid+1, high)
```

Exercise 4.15

We earlier saw the iterative version of the Binary search. Can we write any recursive algorithm as iterative algorithm?

End of Lecture 4

CS213/293 Data Structure and Algorithms 2024

Lecture 5: Tree

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-08-28

Let us study

tree data structure,

which will help us solving many problems including the problem of dictionary.

Commentary: The purpose of programs is to solve problems. We need not invent a data structure until we have a purpose. The purpose will be clarified by the next lecture.

Topic 5.1

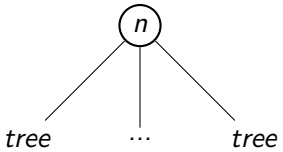
Tree

Tree

Definition 5.1

A tree is either a node

or the following structure consisting of a node and a set of children trees that are disjoint.



The above is our first recursive definition.

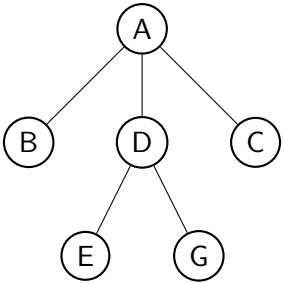
Exercise 5.1

Does the above definition include infinite trees? How would you define an infinite tree?

Example: tree

Example 5.1

An instance of tree.



Some tree terminology(2)

For nodes n_1 and n_2 in a tree T.

Definition 5.2

n_1 is child of n_2 if n_1 is immediately below n_2. We write $n_1 \in children(n_2)$.

Definition 5.3

We say n_2 is parent of n_1 if $n_1 \in children(n_2)$ and write $parent(n_1) = n_2$.
If there is no such n_2, we write $parent(n_1) = \perp$.

Definition 5.4

n_1 is ancestor of n_2 if $n_1 \in parent^*(n_2)$. We write $n_1 \in ancestors(n_2)$.
n_2 is descendant of n_1 if $n_1 \in ancestor(n_2)$. We write $n_1 \in descendants(n_2)$.

Commentary: For a function $f(x)$, we define $f^*(x) = y | y = f(..f(x))$,i.e., the function is applied 0 or more times (informal definition). What would be a mathematically formal definition?

Some tree terminology

Definition 5.5

n_1 and n_2 are siblings if $parent(n_1) = parent(n_2)$.

Definition 5.6

n_1 is a leaf if $children(n_1) = \emptyset$.
n_1 is an internal node if $children(n_1) \neq \emptyset$.

Definition 5.7

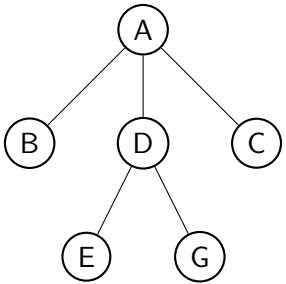
n_1 is a root if $parent(n_1) = Null$.

Exercise 5.2

Can the root be an internal node? Can the root be a leaf?

Example: Tree terminology

Example 5.2



B, D, and C are children of A.

D is the parent of G.

A is an ancestor of G and E is a descendant of A.

A is an ancestor of A.

G and E are siblings.

B, E, G, and C are leaves.

A and D are internal nodes.

A is the root.

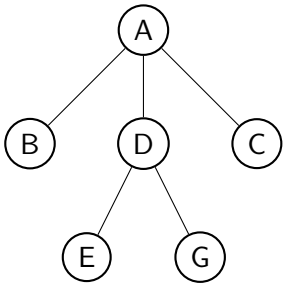
Degree of nodes

Definition 5.8

We define the degree of a node n as follows.

$$degree(n) = |children(n)|$$

Example 5.3



$dgree(A) = 3$

$dgree(B) = 0$

$dgree(D) = 2$

Label of tree

Usually, we store data on the tree nodes.

We define the $label(n)$ of a node n as the data stored on the node.

Level/Depth and height of nodes

Definition 5.9

We define the level/depth of a node n as follows.

$$level(n) = \begin{cases} 0 & \text{if } n \text{ is a root} \\ level(n') + 1 & n' = parent(n) \end{cases}$$

Definition 5.10

We define the height of a node n as follows.

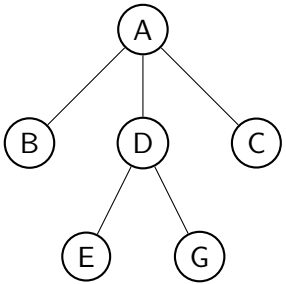
$$height(n) = max(\{height(n') + 1 | n' \in children(n)\} \cup \{0\})$$

Exercise 5.3

Why do we need to take a union with 0 in the definition of height?

Example: Level(Depth) and height of nodes

Example 5.4



$level(A) = 0$

$level(B) = 1$

$level(E) = 2$

$height(E) = 0$

$height(D) = 1$

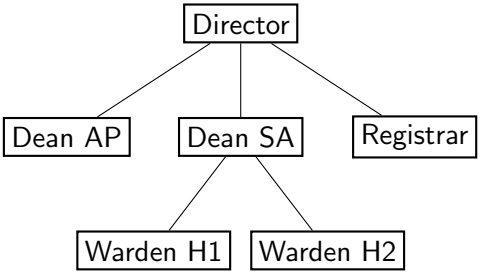
$$height(A) = max(\{height(B) + 1, height(D) + 1, height(C) + 1\} \cup \{0\}) = max(\{1, 2, 1\} \cup \{0\}) = 2$$

Why do we need trees?

A tree represents a hierarchy.

Example 5.5

- *Organization structure of an organization*

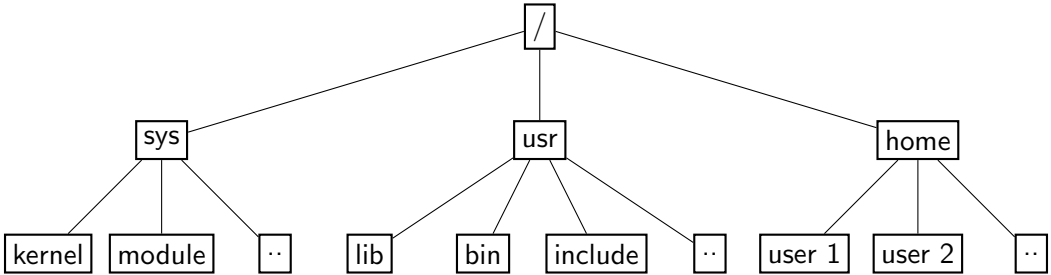


Example: File system

Files are stored in trees in Linux/Windows.

Example 5.6

Part of a Linux file system.



Topic 5.2

Binary tree

Ordered tree

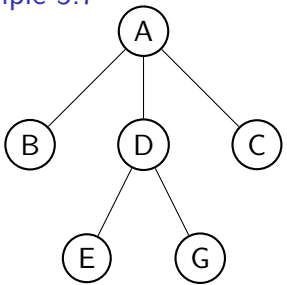
Definition 5.11

A tree is an ordered tree if we assign an order among children.

Definition 5.12

Let n be a node. In an ordered tree, children(n) is a list instead of a set.

Example 5.7



In a tree, we define the children as follows.
$children(A) = \{B, D, C\}$

In an ordered tree, we define the children as follows.
$children(A) = [B, D, C]$

Binary tree

Definition 5.13

An ordered tree T is a binary tree if $|children(n)| \leq 2$ for each $n \in T$.

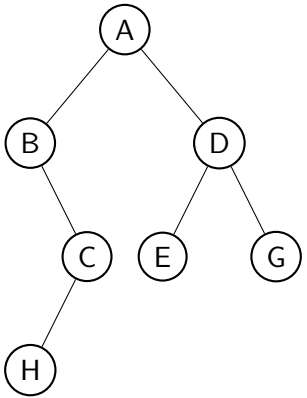
We define the left and right child of n as follows.

- *if $children(n) = [n_1, n_2]$,*
  - *$left(n) = n_1$ and $right(n) = n_2$.*
- *If $children(n) = [n_1]$, n_1 is either left or right child.*
  - *$left(n) = n_1$ and $right(n) = Null$, or*
  - *$left(n) = Null$ and $right(n) = n_1$.*
- *If $children(n) = []$,*
  - *$left(n) = Null$ and $right(n) = Null$.*

Commentary: For a mathematical nerd, the given definition of left/right child is not satisfactory. How can we interpret $children(n) = [n_1]$ in two possible ways? There is an alternative way to define the binary tree. We may say that there are "Null" nodes, which are the leaves. By definition, all internal nodes will have two children. $children(n) = [n_1]$ will be written as either $children(n) = [Null, n_1]$ or $children(n) = [n_1, Null]$. Hence, we will have a clean definition of left and right child. For $children(n) = []$, we will write $children(n) = [Null, Null]$. This issue will come up again in Red-Black tree. Meanwhile, we will stick to our definition.

Example: binary tree

Example 5.8

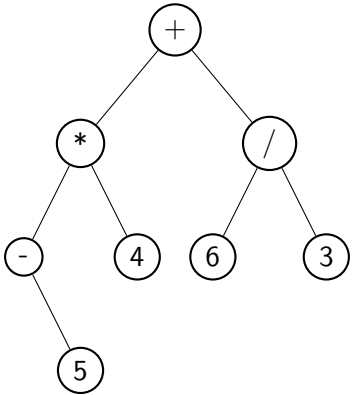


E is the left and G is the right child of D. C is the right child of B. B has no left child.

Usage of binary tree: representing expressions

Example 5.9

Representing mathematical expressions



Exercise 5.4

a. Why do we need an ordered tree?

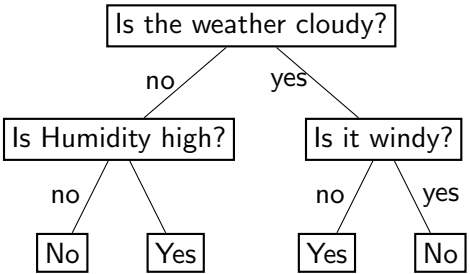
b. How would you evaluate a mathematical expression given as a binary tree?

Usage of binary tree: decision trees in AI

Example 5.10

Does one want to play given the weather?

Given the behavior, we may learn the following tree.



Complete binary tree (*aka* perfect binary tree)

Commentary: Some sources define the complete trees differently, where they allow last level to be not filled and all nodes are as left as possible. They also define full and balanced trees. Do an internet search.

Definition 5.14

A binary tree is complete if the height of the root is h and every level $i \leq h$ has 2^i nodes.

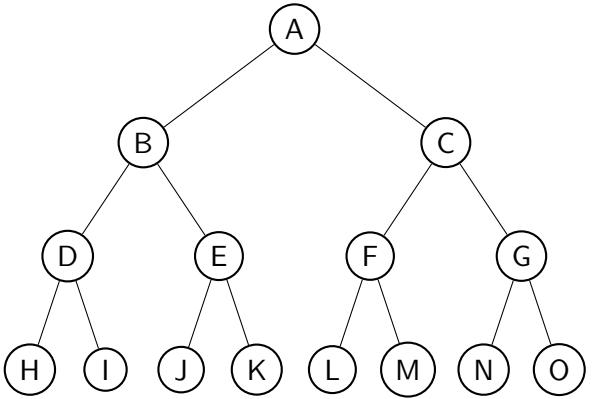
Leaves are only at level h.

The number of leaves $= 2^h$.

Number of internal nodes $= 1 + 2 + ... + 2^{h-1}$
$= 2^h - 1$.

The total number of nodes is $2^{h+1} - 1$.

Example 5.11



Exercise 5.5

a. Prove/Disprove: if no node in the binary tree has a single child, the binary tree is complete.

b. What fraction of nodes are leaves in a complete binary tree?

Maximum and minimum height of a binary tree

Exercise 5.6

Let us suppose there are n nodes in a binary tree.

- *What is the minimum height of the tree?*
- *What is the maximum height of the tree?*

Commentary: For a given height h, a complete binary tree has $2^{h+1} - 1$ nodes. All other binary trees with the height h have fewer nodes. Therefore, $n \leq 2^{h+1} - 1$. Therefore, $log_2 \frac{n+1}{2} \leq h$. The maximum possible height for n nodes is $n - 1$. Therefore, $log_2 \frac{n+1}{2} \leq h \leq n - 1$.

Leaves of binary tree

Theorem 5.1

For a binary tree, $|leaves| \leq 1 + |internal\ nodes|$.

Proof.

We will prove the theorem by induction over the structure of a tree (Recall the recursive definition of a tree).

Base case:

We have a single node.
$|leaves| = 1$ and $|\text{internal nodes}| = 0$. Case holds. ...

Commentary: $|A|$ indicates the size of set A.

Leaves of binary tree(2)

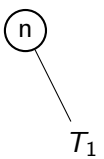
Proof(continued).

Induction step:

We have two cases in the induction step: Root has one child or two children.

Case 1:

Let tree T be constructed as follows.



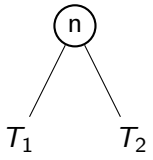
For T_1, let $|leaves| = \ell_1$ and $|\text{internal nodes}| = i_1$.
T has ℓ_1 leaves and $i_1 + 1$ internal nodes.
By the induction hypothesis, $\ell_1 \leq 1 + i_1$.
Therefore, $\ell_1 \leq 1 + i_1 + 1$.
Therefore, $\ell_1 \leq 1 + (i_1 + 1)$. Case holds.

...

Leaves of binary tree(3)

Proof(continued).

Case 2:
Let tree T be constructed as follows.



For T_1, let $|leaves| = \ell_1$ and $|\text{internal nodes}| = i_1$.
For T_2, let $|leaves| = \ell_2$ and $|\text{internal nodes}| = i_2$.
T has $\ell_1 + \ell_2$ leaves and $i_1 + i_2 + 1$ internal nodes.
By induction hypothesis, $\ell_1 \leq 1 + i_1$ and $\ell_2 \leq 1 + i_2$.
Therefore, we have $\ell_1 + \ell_2 \leq 2 + i_1 + i_2$.
Therefore, $\ell_1 + \ell_2 \leq 1 + (i_1 + i_2 + 1)$. Case holds.

□

Exercise 5.7

Prove/Disprove: If no node in the binary tree has a single child, $|leaves| = 1 + |internal\ nodes|$. (Quiz 2023)

Maximum and minimum number of leaves

Let n be the number of nodes in a binary tree T.

Due to the previous theorem, we know $|\text{leaves}| \leq 1 + |\text{internal nodes}|$.

Since $|\text{leaves}| + |\text{internal nodes}| = n$, $|\text{leaves}| \leq 1 + n - |\text{leaves}|$.

$$|\text{leaves}| \leq \frac{(n+1)}{2}.$$

Exercise 5.8

a. When do $|leaves|$ meet the inequality?
b. When is the number of leaves minimum?

Commentary: If T is complete, the number of leaves is $\frac{(n+1)}{2}$.

Topic 5.3

Representing Tree

Container for tree

There is no C++ container for the tree.

Trees are the backbone of many abstract data structures.

For some reason, it is not explicitly there.

Exercise 5.9

Why is there no tree container in C++ STL? (Let us ask ChatGPT)

Commentary: I guess that we rarely explicitly need trees in our programming. We usually have higher goals such as stack,queue, set, and map, which may need a tree as an internal data structure, but users need not be exposed. However, there are applications where there is a clear need for trees. For example, the representation of arithmetic expressions. In my programming, whenever I needed a tree. I have implemented it myself.

Representation of a binary tree on a computer

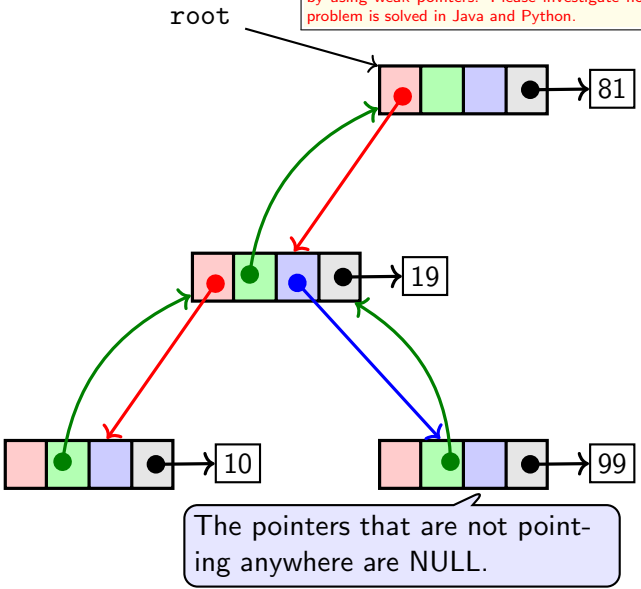
Commentary: parent pointer causes cycle of pointers, which makes the garbage collection difficult in the languages like Java. In C++, we may break the cycle by using weak pointers. Please investigate how the problem is solved in Java and Python.

Definition 5.15

A binary tree consists of nodes containing four pointer fields.

- *left child*
- *parent*
- *right child*
- *label*

An additional root pointer points to the root of the tree.



Exercise 5.10

Do we need the parent pointer?

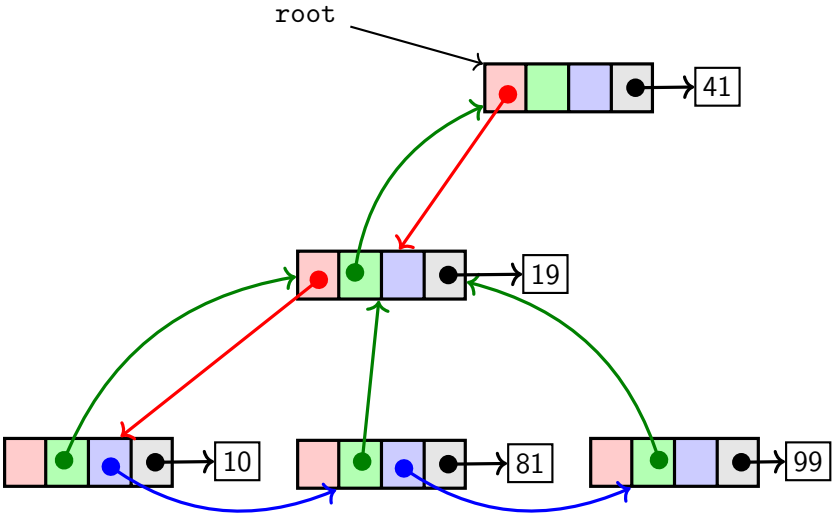
Representation of a tree on a computer

Definition 5.16

A tree consists of nodes containing four pointer fields.

- *first child*
- *parent*
- *next sibling*
- *label*

An additional root pointer points to the root of the tree.



Exercise 5.11

Are we representing an ordered tree or an unordered tree?

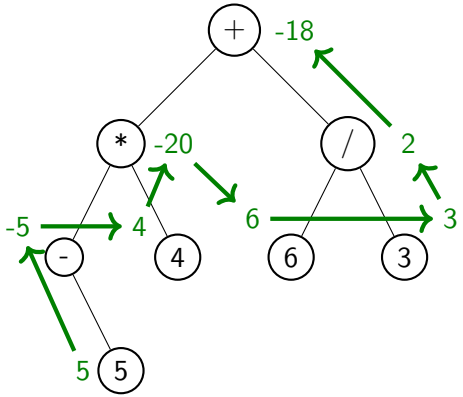
Topic 5.4

Tree walks

Application : Evaluating an expression

Example 5.12

If we want to evaluate an expression represented as a binary tree, we need to visit each node and evaluate the expression in a certain order.



In green, we have evaluated the value of the node. The path indicates the order of evaluation.

Tree walks

Visiting nodes of a tree in a certain order are called tree walks.

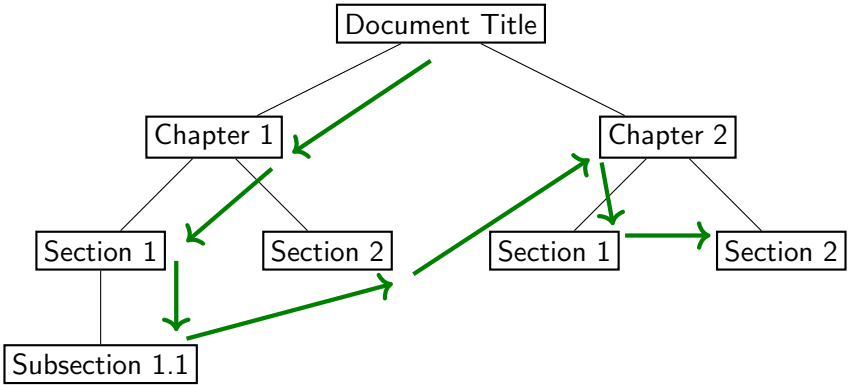
There are two kinds of walks for trees.

- preorder: visit parent first
- postorder: visit children first

Example: preorder

Example 5.13

Let a document be stored as a tree. We read the document in preorder.



Preorder/Postorder walk

Algorithm 5.1: PreOrderWalk(n)

```
1 visit(n);
2 for n' ∈ children(n) do
3     PreOrderWalk(n');
```

Algorithm 5.2: PostOrderWalk(n)

```
1 for n' ∈ children(n) do
2     PostOrderWalk(n');
3 visit(n);
```

The first example of expression evaluation is postorder walk.

Commentary: visit(v) is some action taken during the walk.

Walking on ordered tree

How do we walk on an ordered tree?

For an ordered tree, we may visit children in the given order among siblings.

We may have choices to change the order of visits among ordered siblings.

Commentary: Our algorithm works for both ordered and unordered trees. Our algorithm does not specify the order of visits of siblings for unordered trees. Please pay attention to the subtle differences among trees, ordered trees, and binary trees.

Topic 5.5

Walking binary trees

Preorder/Postorder walk over binary trees

We have more structure in binary trees. Let us write the algorithm for walks again.

Algorithm 5.3: PreOrderWalk(n)

```
1 if n == Null then
2     return
3 visit(n);
4 PreOrderWalk(left(n));
5 PreOrderWalk(right(n));
```

Algorithm 5.4: PostOrderWalk(n)

```
1 if n == Null then
2     return
3 PostOrderWalk(left(n));
4 PostOrderWalk(right(n));
5 visit(n);
```

Exercise 5.12

Are the above programs tail-recursive?

Inorder walk of binary trees

Definition 5.17

In an inorder walk of a binary tree, we visit the node after visiting the left subtree and before visiting the right subtree.

Algorithm 5.5: InOrderWalk(n)

```
1 if n == Null then
2     return
3 InOrderWalk(left(n));
4 visit(n);
5 InOrderWalk(right(n));
```

Exercise 5.13

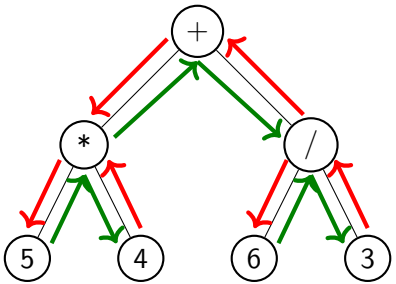
Given complete binary trees with 7 nodes, label the nodes such that the preorder, inorder, or postorder walks produce the sequence 1,2,...,7.

Application : Printing an expression

To print an expression (without unary minus), we need to visit the nodes in inorder.

Algorithm 5.6: PrintExpression(n)

```
1 if n is leaf then
2 |   print(label(n));
3 |   return
4 print("(");
5 PrintExpression(left(n));
6 print(label(n));
7 PrintExpression(right(n));
8 print(")");
```



((5 * 4) + (6 / 3))

Exercise 5.14

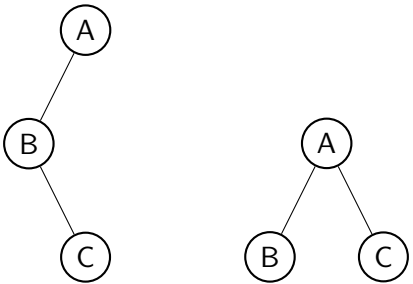
a. Modify the above algorithm to support unary minus.

*b. What will happen if "***if***" at line 1 is replaced by "***if*** n == NULL* **then return"?**

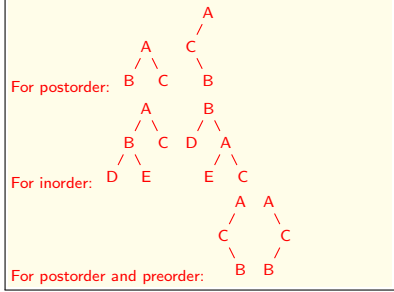
Commentary: The order of the walk is the pattern of recursive calls and actions on nodes. An application may need a mixed action pattern. In the above printing example, we need to print parentheses before and after making recursive calls. The parentheses are printed pre/post-order. All three walks are present in the above algorithm.

Many trees have the same walks

The following two ordered trees have the same preorder walks.



Commentary: Answer:



Exercise 5.15

a. Give two binary trees that have the same postorder walks.

b. Give two binary trees that have the same inorder walks.

c. Give two binary trees that have the same postorder and preorder walks.

Topic 5.6

Tutorial problems

Exercise: paths in a tree

Exercise 5.16

Given a tree with a maximum number of children as k. We give a label between 0 and k-1 to each node with the following simple rules. (i) the root is labeled 0. (ii) For any vertex v, suppose that it has r children, then arbitrarily label the children as 0,..,r-1. This completes the labeling. For such a labeled tree T, and a vertex v, let seq(v) be the labels of the vertices of the path from the root to v. Let $Seq(T) = \{seq(w) | w in T\}$ *be the set of label sequences. What properties does Seq(T) have? If a word w appears what words are guaranteed to appear in Seq(T)? How many times does a word w appear as a prefix of some words in Seq(T)?*

Commentary: From Milind's notes!

Lowest common ancestor(LCA)

Definition 5.18

For two nodes n_1 and n_2 in a tree T, $LCA(n_1, n_2, T)$ is a node in $ancestors(n_1) \cap ancestors(n_2)$ that has the largest level.

Exercise 5.17

Write a function that returns $lca(v, w, T)$. What is the time complexity of the program?

Exercise: paths in a tree

Exercise 5.18

Given $n \in T$, Let $f(n)$ be a vector, where $f(n)[i]$ is the number of nodes at depth i from n.

- *Give a recursive equation for $f(n)$.*
- *Give a pseudo code to compute the vector $f(root(T))$. How is the time complexity of the program?*

The uniqueness of walks if two walks are the same.

Exercise 5.19

Give an algorithm for reconstructing a binary tree if we have the preorder and inorder walks.

Exercise 5.20

Let us suppose all internal nodes of a binary tree have two children. Give an algorithm for reconstructing the binary tree if we have the preorder and postorder walks.

Topic 5.7

Problems

Exercise: mean level

Exercise 5.21

a. Suppose that you are given a binary tree, where, for any node v, the number of children is no more than 2. We want to compute the mean of ht(v), i.e., the mean level of nodes in T. Write a program to compute the mean level.

b. Suppose that we are given the level of all leaves in the tree. Can we compute the mean height? Given a sequence $(n_1,n_2,...,n_k)$ of the levels of k leaves, is there a binary tree with exactly k leaves at the given levels?

Reconstructing tree from preorder walks

Exercise 5.22

Let us suppose we can calculate the number of children of a node by looking at the label of a node of a binary tree, e.g., arithmetic expressions. Give an algorithm for reconstructing the binary tree if we have the preorder walk.

Exercise: previous print

Exercise 5.23

For a given binary tree, let prevPrint(T, a) give the node n' such that label(n') will appear just before label(n) in the inorder printing of T. Give a program that implements prevPrint.

Exercise: level-order walk

Exercise 5.24

Give an algorithm for walking a tree such that nodes are visited in the order of their level. Two nodes at the same level can visit in any order.

Exercise 5.25

Give an algorithm for walking a tree such that nodes are visited in the order of their height.

Exercise: balanced trees

Definition 5.19

A binary tree T is called balanced if for each node $n \in T$

$$|height(right(n)) - height(left(n))| \leq 1.$$

Exercise 5.26

Prove/Disprove: if no node in the binary tree has a single child, the binary tree is balanced.

End of Lecture 5

CS213/293 Data Structure and Algorithms 2024

Lecture 6: Binary search tree (BST)

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-08-25

Ordered dictionary

Recall: There are two kinds of dictionaries.

- Dictionaries with unordered keys
  - We use hash tables to store dictionaries for unordered keys.
- Dictionaries with ordered keys
  - Let us discuss the efficient implementations for them.

Recall: Dictionaries via ordered keys on arrays

- Searching is $O(log\ n)$
- Insertion and deletion is $O(n)$
  - Need to shift elements before insertion/after deletion

Can we do better?

Topic 6.1

Binary search trees

Binary search trees (BST)

Definition 6.1

A binary search tree is a binary tree T such that for each $n \in T$

- *n is labeled with a key-value pair of some dictionary,*
  - *(if $label(n) = (k, v)$, we write $key(n) = k$)*
- *for each $n' \in descendants(\ left(n)\ )$, $key(n') \leq key(n)$, and*
- *for each $n' \in descendants(\ right(n)\ )$, $key(n') \geq key(n)$.*

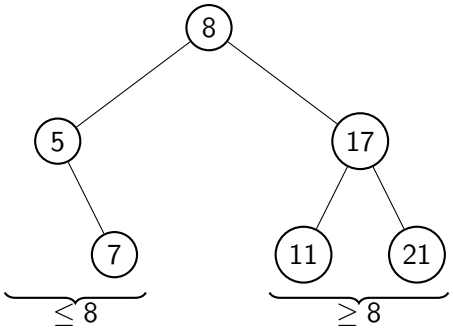
Note that we allow two entries to have the same keys. The same key can be in either of the subtrees.

Commentary: We assume $descendants(Null) = \emptyset$.

Example: BST

Example 6.1

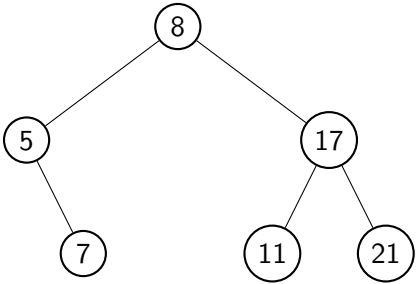
In the following BST, we show only keys stored at the node.

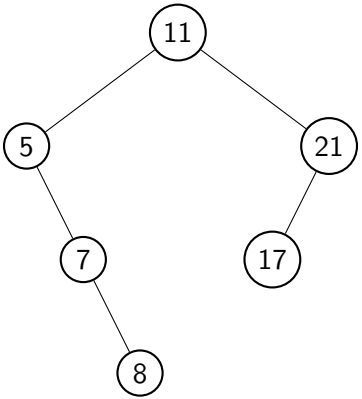


Example: many BSTs for the same data

Example 6.2

The same set of keys may result in different BSTs.

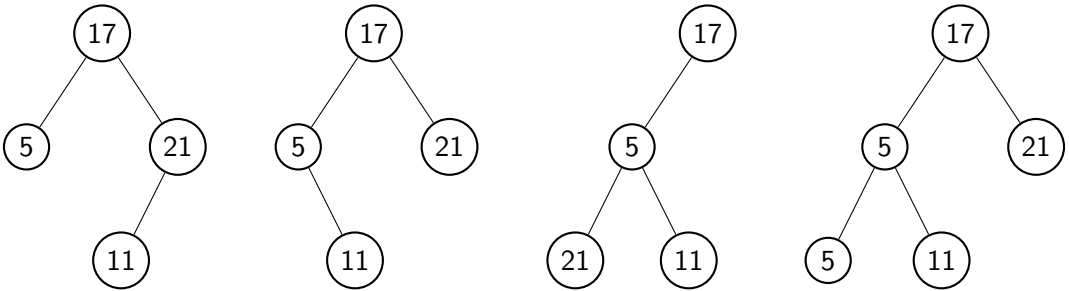




Exercise: Identify BST

Exercise 6.1

Which of the following are BSTs?



Topic 6.2

Algorithms for BST

Algorithms for BST

We need the following methods on BSTs

- search
- insert
- minimum/maximum
- successor/predecessor: Find the successor/predecessor key stored in the dictionary
- delete

Exercise 6.2

Give minimum and successor algorithms for sorted array-based implementation of a dictionary.

Commentary: We did not discuss algorithms for minimum and successor in our earlier discussion of unordered dictionaries. However, we need them for other operations on BST.

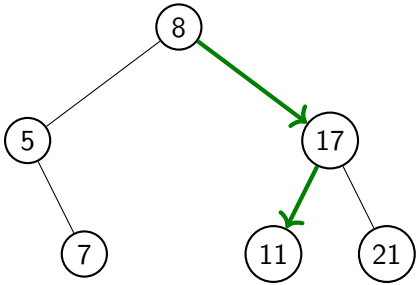
Searching in BST

Commentary: By the definition of BST, we are guaranteed that 11 will not occur in the left subtree of 8. This is the same reasoning as the binary search that we discussed earlier.

Example 6.3

Searching 11 in the following BST.

- We start at the root, which is node 8
- At node 8, go to the right child because $11 > 8$.
- At node 17, go to the left child because $11 < 17$.
- We find 11 at the node.

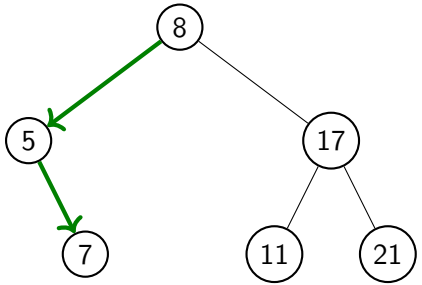


Unsuccessful search in BST

Example 6.4

Searching 6 in the following BST.

- We start at the root, which is node 8
- At node 8, go to the left child because $6 < 8$.
- At node 5, go to the right child because $6 > 5$.
- At node 7, go to the left child because $6 < 7$.
- Since node 7 has no left child the search fails.



Algorithm: Search in BST

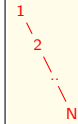
Algorithm 6.1: SEARCH(BST T, int k)

```
1 n := root(T);
2 while n ≠ Null do
3     if key(n) = k then
4         break
5     if key(n) > k then
6         n := left(n)
7     else
8         n := right(n)
9 return n
```

- Running time is $O(h)$, where h is height of BST.
- If there are n keys in the BST, the worst case running time is $O(n)$.

Commentary: Answer:
a. We search in the BST. If the key is found on a node, then we start two(Why?) searches in both the subtrees of the found node. We recursively start the searches.

b. Find N in the following BST



Exercise 6.3

a. Modify the above algorithm to find all occurrences of key k.

b. Give an input of SEARCH *that exhibits worst-case running time.*

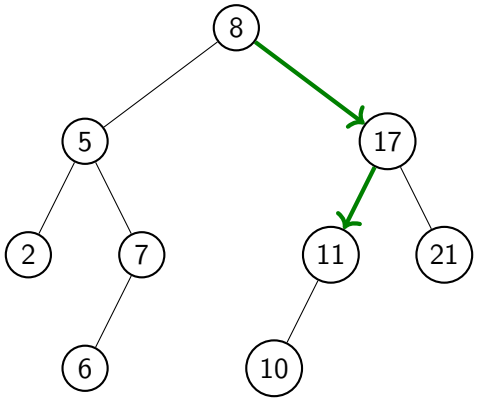
Topic 6.3

Insert in BST

Example: Insert in BST

Example 6.5

Where do we insert 10?



Algorithm: Insert in BST

Algorithm 6.2: INSERT(BST T, Node n)

```
1  x := root(T); y := Null;
2  while x ≠ Null do
3  |   y := x;
4  |   if key(x) > key(n) then
5  |   |   x := left(x)
6  |   else
7  |   |   x := right(x)
8  if y = Null then
9  |   root(T) = n;
10 if key(y) > key(n) then
11 |   left(y) := n
12 else
13 |   right(y) := n
14 parent(n) = y
```

Exercise 6.4

a. *What is the running time of the algorithm?*
b. *Give an order of insertion for the maximum tree height.*
c. *Give an order of insertion for the minimum tree height.*
d. *What happens if key(n) already exists?*

Commentary: Answer:
a. the same as search,
b. 1,2,3,4,5,...,n
c. n/2,n/4,3n/4,n/8,3n/8,5n/8,7n/8,..
d. This algorithm always goes right. It is correct but may not be a good idea. It should randomly choose left or right.

Topic 6.4

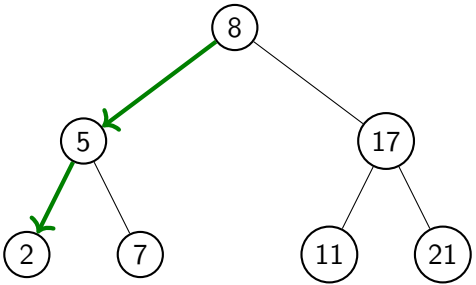
Minimum in BST

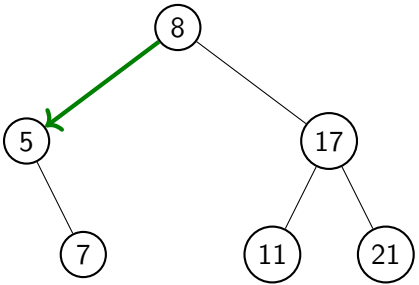
Example: minimum in BST

Commentary: Always go left to find a smaller node. As soon as we do not have a left child, we have found the minimum node.

Example 6.6

What is the minimum of the following BSTs?





Algorithm: Minimum in BST

The following algorithm computes the minimum in the subtree rooted at node n.

Algorithm 6.3: MINIMUM(Node n)

```
1 while n ≠ Null and left(n) ≠ Null do
2     n := left(n)
3 return n
```

- Runtime analysis is the same as SEARCH.

Exercise 6.5

Modify the above algorithm to compute the maximum

Correctness of MINIMUM

Commentary: Note that $key(n') \leq key(n) \leq key(n'')$ where $n'' \in descendants(right(n))$

Theorem 6.1

If $n \neq Null$, the returned node by MINIMUM(n) *has the minimum key in the subtree rooted at n.*

Proof.

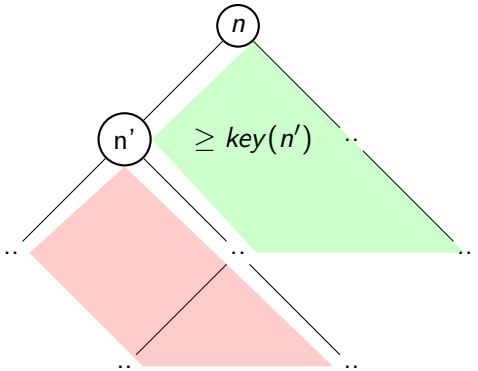
If $left(n) = Null$, $key(n)$ is the minimum key.

Otherwise, we go to $n' = left(n)$. Any node not in $descendants(n')$ must have a larger key than $key(n')$.(Why?)

So the minimum of $descendants(n')$ is the overall minimum.

This argument continues to hold for any number of iterations of the loop. (induction)

Therefore, our algorithm will compute the minimum.



Topic 6.5

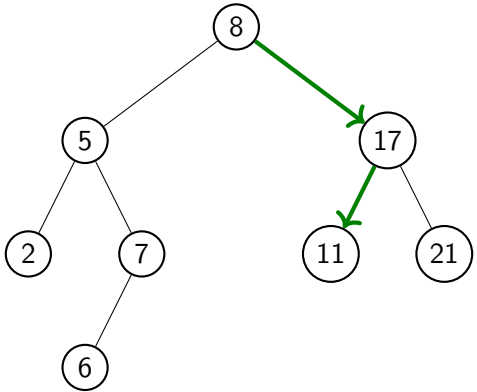
Successor in BST

Example: successor in BST

We now consider the problem of finding the node that has the successor key of a given node.

Example 6.7

Where is the successor of 8?

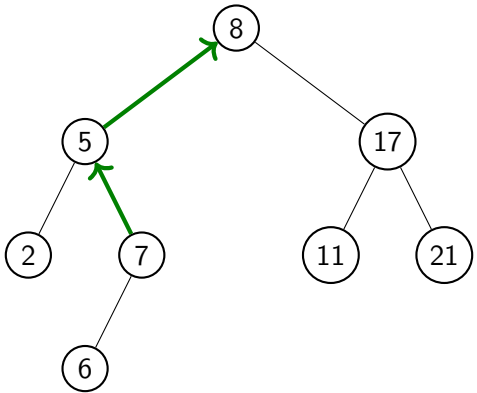


Observation: Minimum of right subtree.

Example: successor in BST(2)

Example 6.8

Where is the successor of 7?



Exercise 6.6

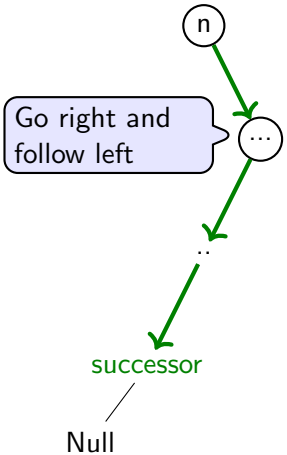
a. When do we not have the successor in the right subtree?

b. If the successor is not in the right subtree, where else can it be?

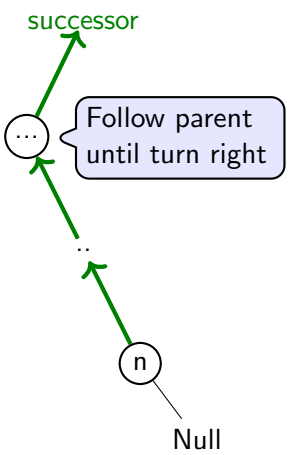
Successor locations

Finding a successor to n

Case 1: If there is a right subtree:



Case 2: If there is no right subtree:



Successor in BST

Algorithm 6.4: SUCCESSOR(BST T, node n)

if $right(n) \neq Null$ **then**
  return MINIMUM($right(n)$)

while $parent(n) \neq Null$ *and* $right(parent(n)) = n$ **do**
  $n := parent(n)$;

return $parent(n)$

Exercise 6.7

a. Modify the above algorithm to compute predecessor
b. What is the running time complexity of SUCCESSOR*?*
c. What happens when we do not have any successor?
d. What is returned if multiple keys have the same value?

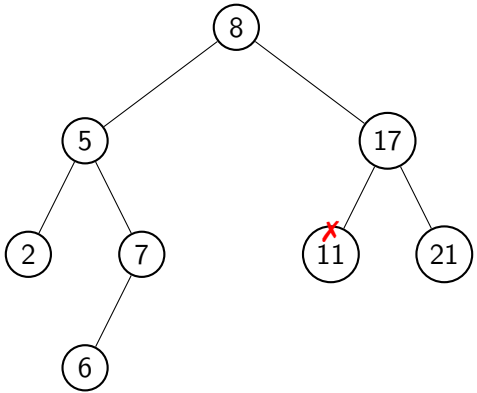
Topic 6.6

Deletion

Example: deleting a leaf

Example 6.9

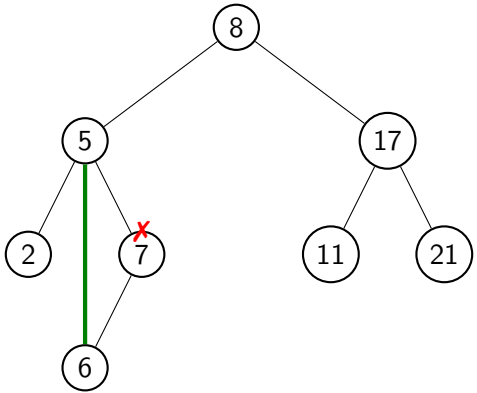
We delete leaf 11 *by simply removing the node.*



Example: deleting a node with a single child

Example 6.10

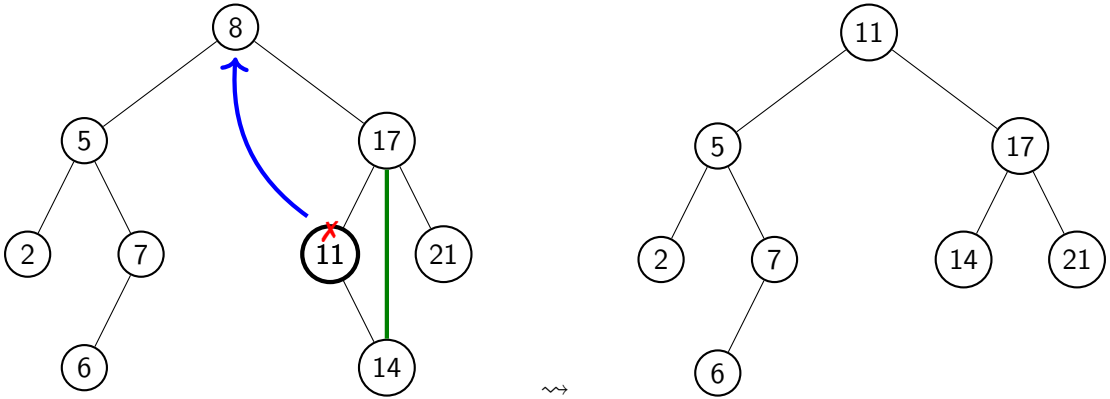
We delete node 7 by making 6 child of 5 and removing the node.



Example: deleting a node with both children

Example 6.11

We delete node 8 *by removing* 11, *which is the successor of* 8, *and then storing the data of* 11 *on* 8.



Algorithm: delete in BST

Algorithm 6.5: DELETE(BST T, Node n)

$y := (left(n) = Null \vee right(n) = Null)\ ?\ n\ :\ \text{SUCCESSOR}(T, n)$; // y will be deleted

if $y \neq n$ **then**

  $key(n) := key(y)$ // copy all data on y

$x := (left(y) = Null)\ ?\ right(y)\ :\ left(y)$; //$x$ is the child of y or x is Null

if $x \neq Null$ **then**

  $parent(x) = parent(y)$ //y is not a leaf, update the parent of x

if $parent(y) = Null$ **then**

  $root(T) = x$ // y was the root, therefore x is root now

else

  if $left(parent(y)) = y$ **then**

    $left(parent(y)) := x$ //Remove y from the tree

  else

    $right(parent(y)) := x$ //Remove y from the tree

Topic 6.7

Average BST depth

Average cost of n-inserts

Let us consider a random permutation of $1, .., n$.

We insert the numbers in the order.

The total cost of insertions will be the sum of the levels of nodes in the resulting BST.

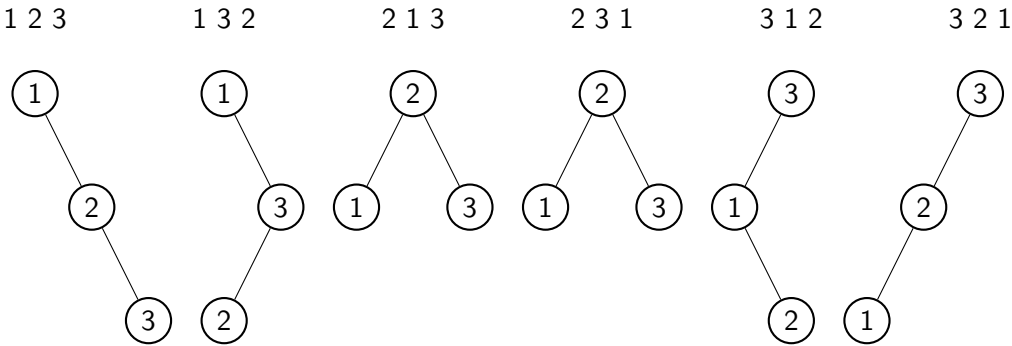
Definition 6.2
Let $T(n)$ denote the average time taken over $n!$ permutations to insert n keys.

Exercise 6.8
What are the best and worst insertion times?

Example: Computing $T(n)$

Example 6.12

Let us compute the average cost of inserting three elements.



$T(3) = 16/6$

Recurrence for $T(n)$

In $(n-1)!$ permutations, i is the first element.

In the permutations,

- i is the root,
- keys $1, ..., i-1$ are in the left subtree, and
- keys $i+1, ..., n$ are in the right subtree.

Recurrence for $T(n)(2)$

There are $(i-1)!$ orderings for keys $1, ..., i-1$.

In the $(n-1)!$ permutations, each ordering of $(i-1)!$ occurs $(n-1)!/(i-1)!$.

If we only had keys $1, ..., i-1$, the average time is $T(i-1)$.

The total time to insert in all the orderings is $(i-1)!\,T(i-1)$.

Recurrence for $T(n)(3)$

While inserting keys $1, .., i-1$, each key is compared with root i, which is an additional unit cost per insertion.

Therefore, the total time of insertion of $(i-1)!$ orderings is

$$(i-1)!(T(i-1)+i-1).$$

Since each permutation occurs $(n-1)!/(i-1)!$, total time for insertions in the left subtree is

$$(n-1)!(T(i-1)+i-1).$$

Similarly, the total time for insertions in the right subtree is

$$(n-1)!(T(n-i)+n-i).$$

Recurrence for $T(n)(4)$

The total time to insert all keys in the permutations where the first key is i is

$$(n-1)!(T(i-1)+T(n-i)+n-1).$$

Therefore, the total time of insertions in all permutations

$$(n-1)!\sum_{i=1}^{n}(T(i-1)+T(n-i)+n-1).$$

Recurrence for $T(n)$(5)

Therefore, the average time of insertions in all permutations

$$T(n) = \frac{(n-1)!}{n!} \sum_{i=1}^{n} (T(i-1) + T(n-i) + n - 1).$$

After simplification,

$$T(n) = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n - 1,$$

where $T(0) = 0$.

What is the growth of $T(n)$?

We need to find an approximate upper bound of $T(n)$.

Let us solve the recurrence relation.

Simplify the recurrence relation

The relation for $n-1$.

$$T(n-1) = \frac{2}{n-1}\sum_{i=0}^{n-2} T(i) + n - 2,$$

After reordering the terms.

$$\sum_{i=0}^{n-2} T(i) = \frac{n-1}{2}(T(n-1) - n + 2),$$

After reordering of terms in $T(n)$,

$$T(n) = \frac{2}{n}\sum_{i=0}^{n-2} T(i) + \frac{2}{n}T(n-1) + n - 1 = \frac{n-1}{n}(T(n-1) - n + 2) + \frac{2}{n}T(n-1) + n - 1,$$

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

Approximate recurrence relation

From

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

we can conclude

$$T(n) \leq \frac{n+1}{n} T(n-1) + 2.$$

Expanding the approximate recurrence relation

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

$$
T(n) \leq \frac{n+1}{n-(n-1)} T(0) + \frac{n+1}{2} 2 + \ldots + \frac{n+1}{n} 2 + 2
$$

Expanding the approximate recurrence relation

Commentary: $\int_1^n \frac{1}{x} dx = \ln n$

$$T(n) \leq 2(n+1)(\underbrace{\frac{1}{2} + ... + \frac{1}{n}}_{\leq ln\ n}) + 2$$

$$T(n) \leq 2(n+1)(ln\ n) + 2$$

Therefore,

$$T(n) \in O(nlog\ n)$$

Topic 6.8

Tutorial problems

Exercise: Sorting via BST

Exercise 6.9

a. Show that in order printing of BST nodes produces a sorted sequence of keys.

b. Give a sorting procedure using BST.

c. Give the complexity of the procedure.

Exercise: delete all smaller keys

Exercise 6.10

Given a BST T and a key k, the task is to delete all keys $b < a$ from T. Write pseudocode to do this. How much time does your algorithm take? What is the structure of the tree left behind? What is its root?

Exercise: expected height

Exercise 6.11

Let H(n) be the expected height of the tree obtained by inserting a random permutation of [n]. Write the recurrence relation for H(n).

Exercise: find the leftmost and rightmost

Exercise 6.12

Given a BST tree T and a value v, write a program to locate the leftmost and rightmost occurrence of the value v.

Topic 6.9

Problems

Exercise: post-order search tree

Exercise 6.13

Consider a binary tree with labels such that the postorder traversal of the tree lists the elements in increasing order. Let us call such a tree a post-order search tree. Give algorithms for search, min, max, insert, and delete on this tree.

Exercise: permutations

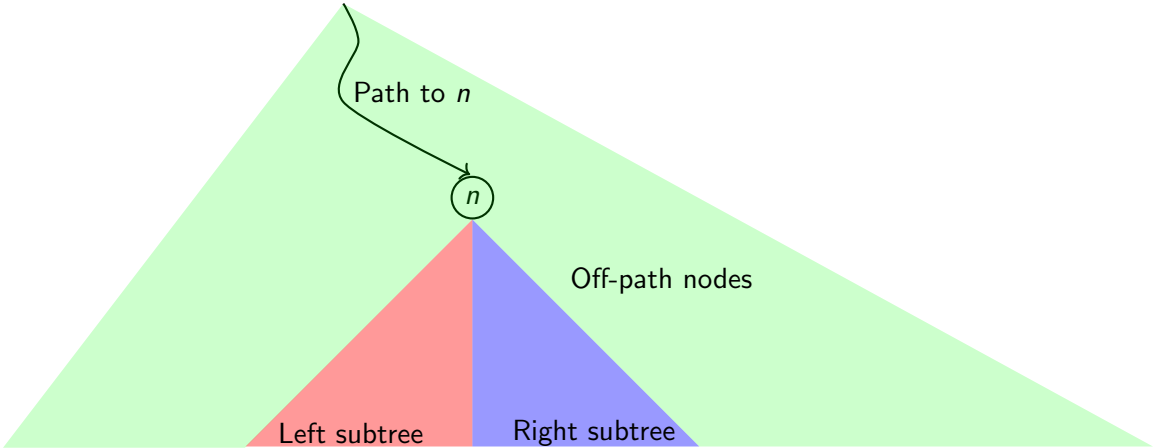
Exercise 6.14

Let [a(1),...,a(n)] be a random permutation of n. Let p(i) be the probability that a(a(1))=i. Compute p(i).

Topic 6.10

Extra slides: proof of correctness of successor

Parts of BST with respect to a node n



The least common ancestor(LCA) is in the middle

Theorem 6.2

For nodes n_1 and n_2, let $n = LCA(n_1, n_2)$. If $key(n_1) \leq key(n_2)$, $key(n_1) \leq key(n) \leq key(n_2)$.

Proof.

We have four cases.

case $n_1 \in ancestors(n_2)$: Trivial.(Why?)

case $n_2 \in ancestors(n_1)$: Trivial.

case $key(n_1) = key(n_2)$:
Since $key(n)$ divided one of the nodes to left and the other to right, $key(n) = key(n_1)$.

case $key(n_1) < key(n_2)$:
n_1 and n_2 must be in the left and right subtree of n respectively.
Therefore, $key(n_1) \leq key(n) \leq key(n_2)$. □

Larger ancestors keep growing!

Theorem 6.3
If $n_2 \in ancestors(n_1)$, $n_1 \in ancestors(n)$, *and* $key(n_2) > key(n)$, *then* $key(n_2) \geq key(n_1)$.

Proof.
n must be in the left subtree of n_2.

n_1 must be in the subtree.(Why?)

Since n_1 is in the left subtree of n_2, $key(n_2) \geq key(n_1)$. □

Correctness of SUCCESSOR

In the following proof, we assume that all nodes have distinct elements.

Theorem 6.4

Let T be a BST, node $n \in T$, and $n' = \text{SUCCESSOR}(n)$.
If $n' \neq Null$, $key(n') > key(n)$ and for each node $n'' \in T - \{n, n'\}$, we have

$$\neg(key(n) < key(n'') < key(n')).$$

Proof.

Claim: A successor of n cannot be an off-path node.
Assume an off-path node n' is the successor of n.
Therefore, $key(n) < key(n')$.
Due to theorem 6.2, $key(n) \leq key(LCA(n, n')) \leq key(n')$.
Therefore, $key(LCA(n, n'))$ is between the nodes. Contradiction. ...

Correctness of SUCCESSOR(2)

Proof(Continued).

Claim: Successor of n cannot be in left subtree.
All nodes will have keys that are less than $key(n)$.

Claim: If the right subtree exists, then a successor cannot be on the path to n.

1. Consider $n' \in descendants(right(n))$.
2. Therefore, $key(n') > key(n)$.
3. For some $n'' \in ancestors(n)$, let us assume n'' is successor of n.
4. Therefore, $key(n'') > key(n)$.
5. Therefore, $n \in descendants(left(n''))$.
6. Therefore, $n' \in descendants(left(n''))$. due to 1 and 5
7. Therefore, $key(n'') > key(n')$.
8. Therefore, $key(n'') > key(n') > key(n)$.
9. Therefore, $key(n'')$ is not a successor. Contradiction. ...

Correctness of SUCCESSOR(2)

Proof(Continued).

Claim: If the right subtree exists, the successor is the minimum of the right subtree.
Since the successor is nowhere else, it must be the minimum.

Claim: If there is no right subtree and there is a node greater than n, the successor is the closest node on the path to n such that the key of the node is greater than n.
Let $n_1, n_2 \in ancestors(n)$ such that $n_2 \in ancestors(n_1)$, $key(n_2) > key(n)$, and $key(n_1) > key(n)$.
Due to theorem 6.3, $key(n_2) > key(n_1)$.
Therefore, n_2 cannot be a successor.
Therefore, the closest node to n is the successor. □

Exercise 6.15

a. Show that the closest node in the above proof must have n in its right subtree.
b. There is a final case missing in the above proof. What is the case? Prove the case.
b. Modify the above proof to support repeated elements in BST.

End of Lecture 6

CS213/293 Data Structure and Algorithms 2024

Lecture 7: Red-Black Trees

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-09-01

Topic 7.1

Balance and rotation

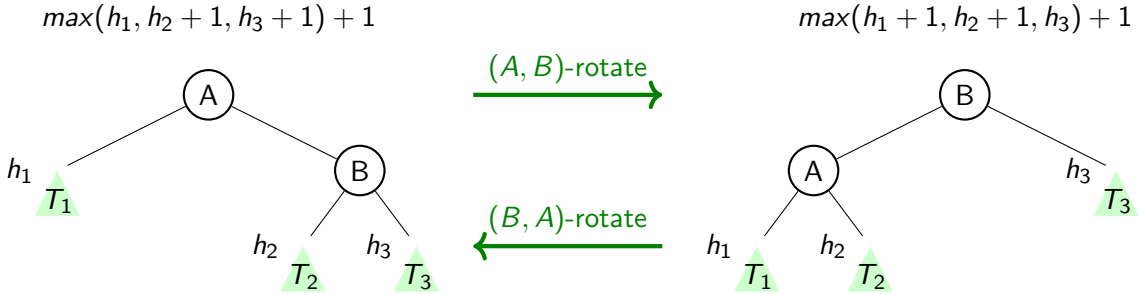
Maintain balance

BST may have a large height, which is bad for the algorithms.

Height is directly related to branching. More branching implies a shorter height.

We call BST imbalanced when the difference between the left and right subtree height is large.

Balancing height by rotation



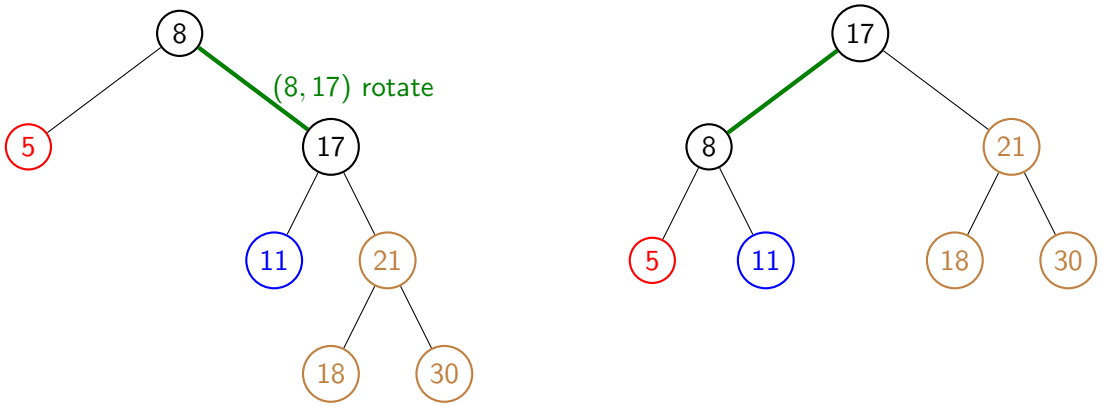
For example, if $h_3 > h_2 = h_1$ and we rotate the BST, we will get a valid and more balanced BST with less height.

Rotation may be applied in the reverse direction, where A is the left child of B. We define the symmetric rotation in both directions.

Example: rotation

Example 7.1

In the following BST, we can rotate 8-17 *edge.*



Commentary: This tree operation is important. Please carefully observe the destination of red,blue, and brown subtrees.

When to rotate? Can only rotation fix imbalance?

Rotation is a local operation, which must be guided by non-local measure height.

We need a definition of balance such that rotations operations should be able to achieve the balance.

Design principle:
We minimize the number of rotations while allowing some imbalance.

Topic 7.2

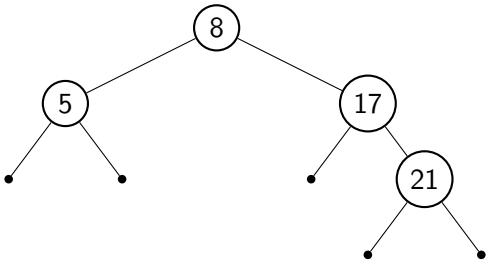
Red-black tree

Null leaves

To describe a red-black tree, we replace the null pointers for absent children with dummy null nodes.

Example 7.2

The following tiny nodes are the dummy null nodes.



Red-black tree

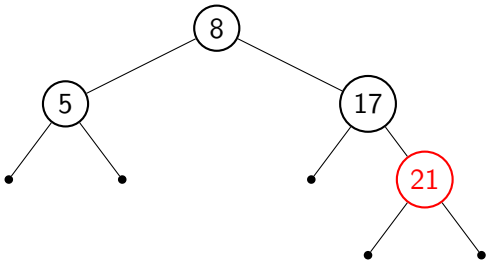
Definition 7.1

A red-black tree T is a binary search tree such that the following holds.

- *All nodes are colored either* **red** *or* **black**
- *Null leaves have no color*
- *Root is colored black*
- *All* **red** *nodes do not have* **red** *children*
- *All paths from the root to null leaves have the same number of* **black** *nodes.*

Example 7.3

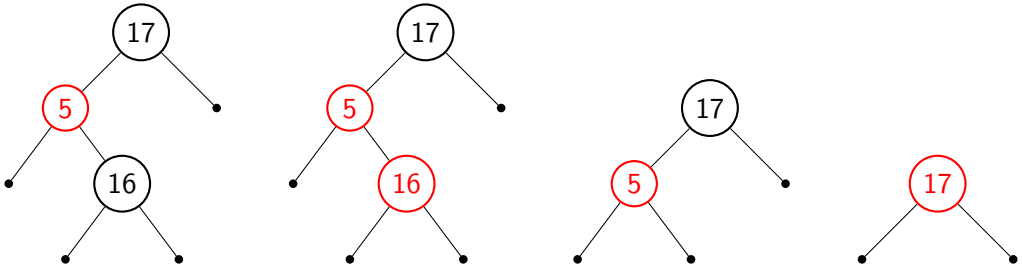
An example of a red-black tree.



Exercise: Identify red-black trees

Exercise 7.1

Which of the following are red-black trees?



Observations:

- Red nodes are not counted in the imbalance. We need them only when there is an imbalance.
- There cannot be too many red nodes. (Why?)
- Red nodes can be at every level except the root.

Black height

Definition 7.2

The black height (bh) for each node is defined as follows.

$$bh(n) = \begin{cases} 0 & n \text{ is a null leaf} \\ max(bh(right(n)), bh(left(n))) + 1 & n \text{ is a } \textbf{black} \text{ node} \\ max(bh(right(n)), bh(left(n))) & n \text{ is a red node} \end{cases}$$

Until it is stated otherwise, all heights mentioned in this lecture are black heights.

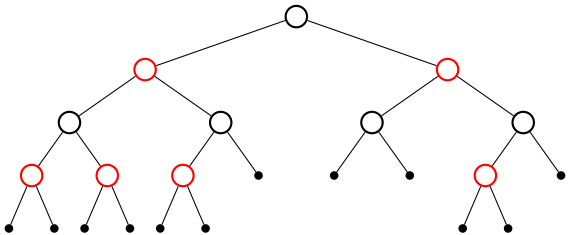
Exercise 7.2

Prove that for each node n in a red-black tree, $bh(right(n)) == bh(left(n))$.

Example: black height

Example 7.4

The black height of the following red-black tree is 2.



Exercise 7.3

Can we change the color of some nodes without breaking the conditions of a red-black tree?

Bound on the height of a red-black tree

Let h be the black height of a red-black tree containing n nodes.

- n is the smallest when all nodes are **black**. Therefore, the tree is a complete binary tree. Therefore, $n = 2^h - 1$.
- n is largest when the alternate levels of the tree are red. The height of the tree is $2h$. Therefore, $n = 2^{2h} - 1$.

$$\log_4 n < h < 1 + \log_2 n$$

Search, Maximum/Minimum, and Successor/Predecessor

We can search, maximum/minimum, and successor/predecessor on the red-black tree as usual.

Their running time will be $O(\log n)$ because $h < 1 + \log_2 n$.

How do we do insertion and deletion on a red-black tree?

Topic 7.3

Insertion in red-black tree

BST insertion in red-black tree

1. Follow the usual algorithm of insertion in the BST, which inserts the new node n as a leaf.
   - Note that there are dummy nodes. n is inserted as the parent of a dummy node.
2. We color n red.

- Good news: No change in the black height of the tree.
- Bad news: n may have a *red* parent.

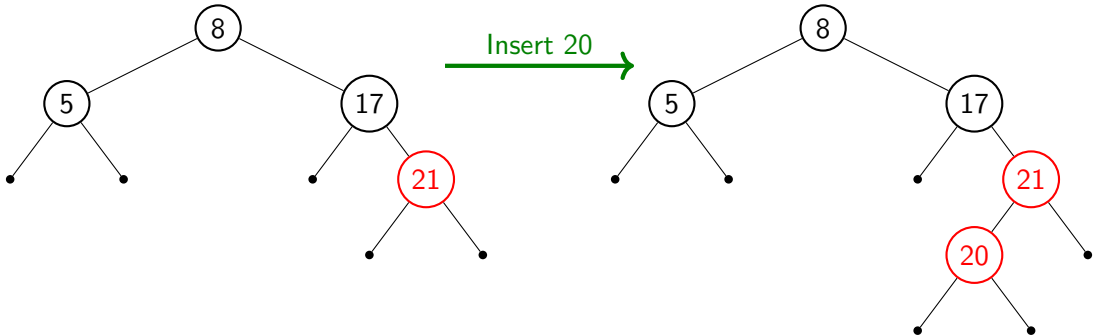
After insertion, we may have a **red**-**red** violation, where a **red** node has a **red** child.

Commentary: We have null nodes in this setting. We need to add nulls as children to the new node.

Example: insert in red-black tree

Example 7.5

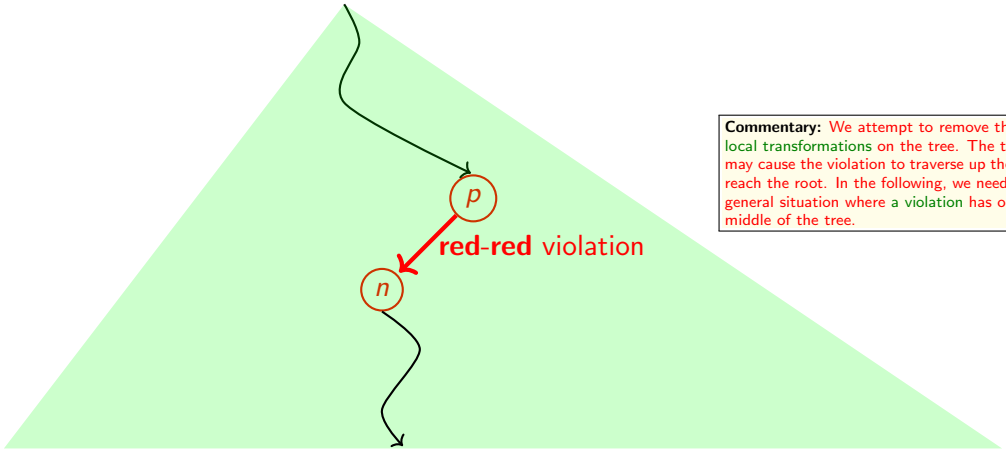
Inserting 20 *in the following tree.*



The insertion results in violation of the condition of the red-black tree, which says red nodes can only have black children.

Iteratively remove **red**-**red** violation

A **red**-**red** violation starts at a leaf. However, an attempt to remove the violation may push up the violation to the parent.

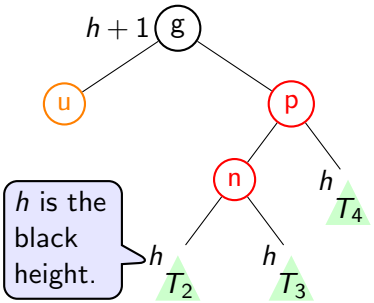


Commentary: We attempt to remove the violation by local transformations on the tree. The transformation may cause the violation to traverse up the tree until we reach the root. In the following, we need to consider a general situation where a violation has occurred in the middle of the tree.

Pattern around red-red violation

Commentary: Since the new node is always added at the leaf, one may think that h must be zero, which may not be the case. As noted in the previous slide, the violation may move upwards.

If n has a **red** parent, we correct the error by rotation and re-coloring operations.



Orange means that we need to consider all possible colors of the nodes.

We have three cases.

- Case 1: u is **red**
- Case 2: u is not **red** and the g to n path is not straight
- Case 3: u is not **red** and the g to n path is straight

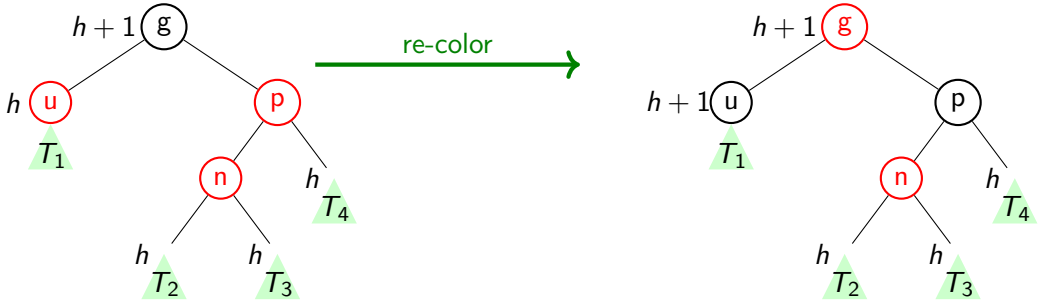
"not **red**" means either **black** node or null node.

Exercise 7.4

Why must g exist and be black?

No transformation should change the black height of g.

Case 1: The uncle is red



In the subtree of g, there is no change in the black height and no **red-red** violation.

Now g is red. We have three possibilities: the parent of g is **black**, the parent of g is **red**, and g is the root.

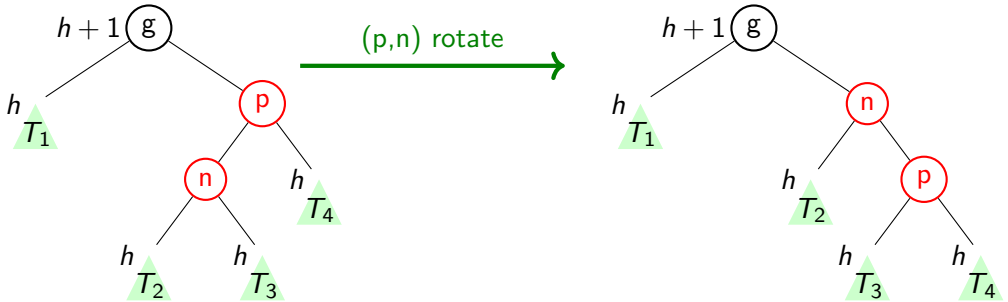
Exercise 7.5

What do we do in each of the possibilities?

Commentary: Possibility 1: Nothing. Possibility 2: We have a red-red violation a level up and need to apply the transformations there. Possibility 3: turn g back to black.

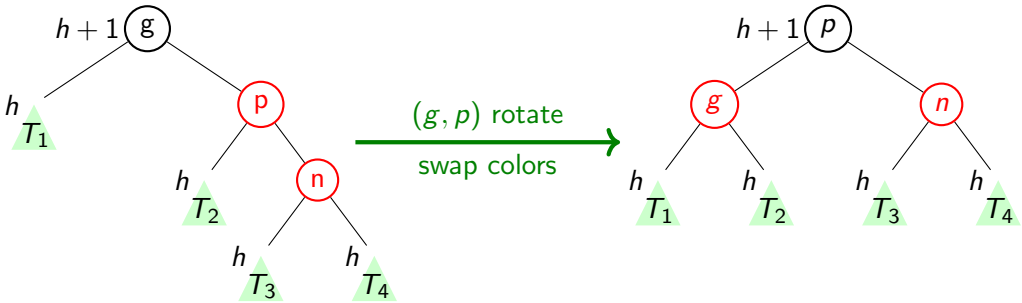
Case 2: The uncle is not red and the path to the grandparent is not straight

straight means $left^2(parent^2(n)) = n$
or $right^2(parent^2(n)) = n$



This transformation does not resolve the violation but converts the violation to case 3.

Case 3: The uncle is not red and the path to the grandparent is straight



The transformation removes the red-red violation.

Exercise 7.6

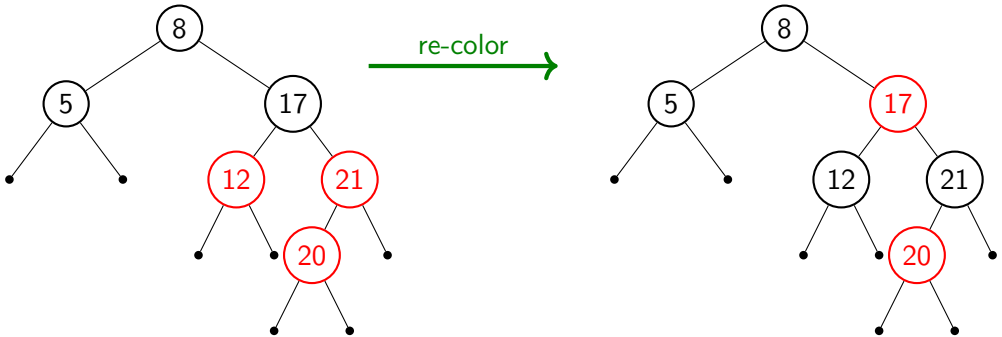
a. Why are the roots of T_2, T_3, and T_4 not **red***?*

b. Show that if the root of T_1 is **red** *then the above operation does not work.*

Example: red-red correction case 1

Example 7.6

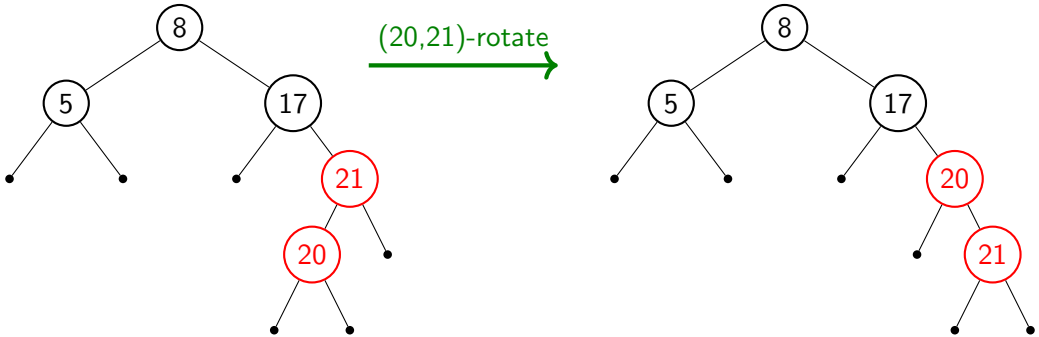
We just inserted 20 *in the following tree. We need to apply case 1 to obtain a red-black tree.*



Example: red-red correction case 2

Example 7.7

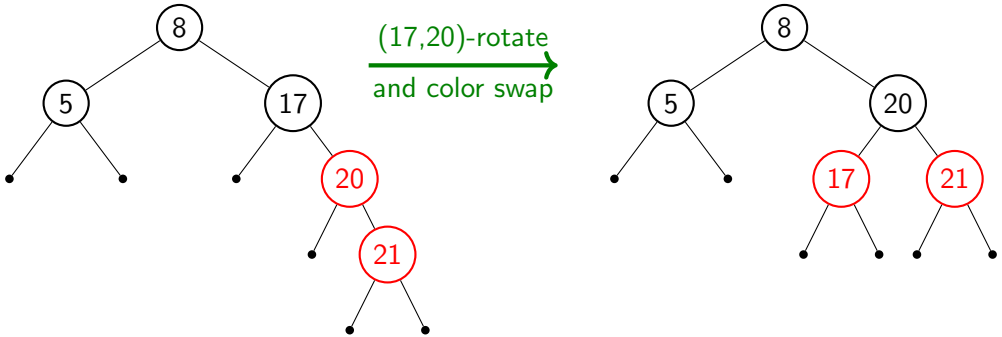
Consider the following example. We are attempting to insert 20. *We apply case 2 to move towards a red-black tree.*



The above is not a red-black tree. Furthermore, we need to apply case 3 to obtain the red-black tree.

Example: red-red correction case 3 (continued)

We apply case three as follows.



Summary of insertion

Insert like BST, make the new node **red**, and run the following algorithm.

Algorithm 7.1: RedRedRepair(Node n)

if *n is root* **then** n.color := black; **return**;
p := parent(n);
if *p is black* **then return**;
g := parent(p); u := sibling(p);
switch(pattern around *n*):
case 1: Re-color according to case 1; RedRedRepair(g);
case 2: Rotate according to case 2; RedRedRepair(p); // p is in case 3
case 3: Re-color and Rotate according to case 3;

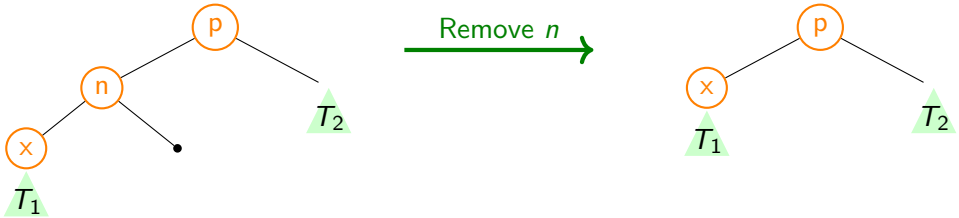
Commentary: What is the intuitive difference between cases 1 and 3? Case 1 reduces the total number of red nodes in a subtree. Case 3 balances if there are too many red nodes on one side of the subtree.

Topic 7.4

Deletion in red-black tree

BST deletion in red-black tree

- Delete a node as if it is a binary search tree.
- Recall: In the BST deletion we always delete a node n with at most one non-null child.



x can be either a null or non-null node.

What can go wrong with a red-black tree?

Since a child x of n takes the role of n, we need to check if x can replace n.

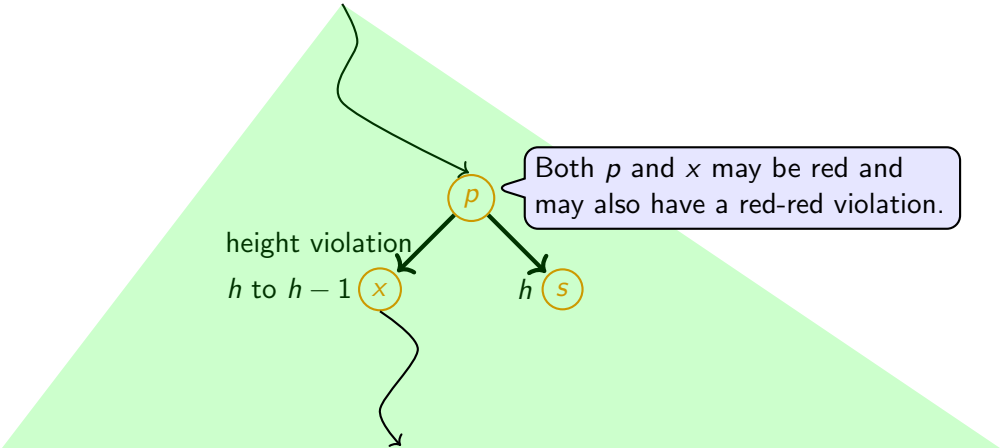
- If n was **red**, no violations occur. (Why?)
- If n was **black**, $\underbrace{bh(x) = bh(n) - 1}_{\text{black height violation}}$, or it is possible that $\underbrace{\text{both } x \text{ and } p \text{ are } \textbf{red}}_{\text{red-red violation}}$.

The leaves of the subtree rooted at x will have one less black depth.

Commentary: The or in the second bullet point is not xor. Both violations are possible at the same time.

Violation removal procedure

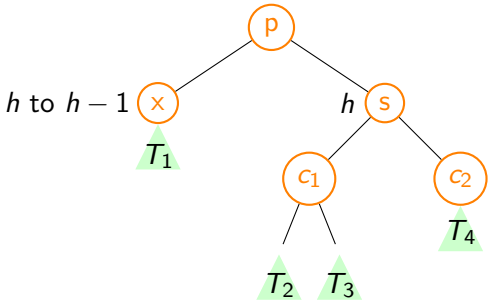
To remove the violation at x, we may recolor and rotate around x, which may push the violation to the parent. Therefore, we consider that the violation is in the middle of the tree.



Orange means that we need to consider all possible colors of the nodes.

Violation pattern

After deletion, we may need to consider the following five nodes around x.



We correct the violation either by rotation or re-coloring.

There are six cases

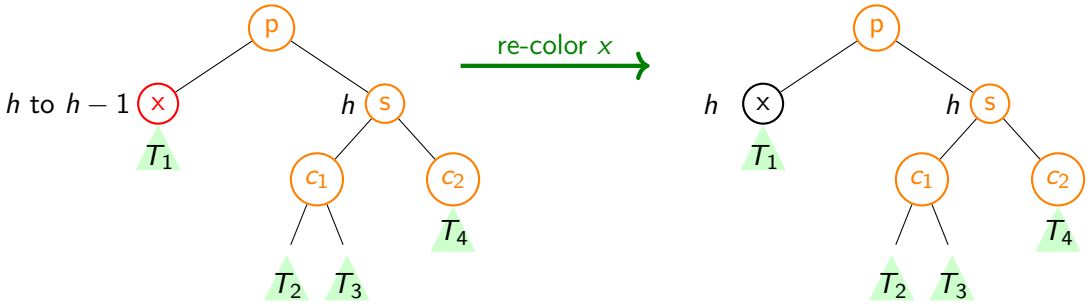
1. x is **red**
2. x is not **red** and root.
3. x is not **red** and s is **red**
4. x is not **red** and s is **black**
   - 4.1 c_2 is **red**
   - 4.2 c_2 is not **red** and c_1 is **red**
   - 4.3 c_2 is not **red** and c_1 is not **red**

The goal is to restore the black height of p.

Exercise 7.7

Show: If x is not root and not **red**, *s must exist.*

Case 1: x is **red**

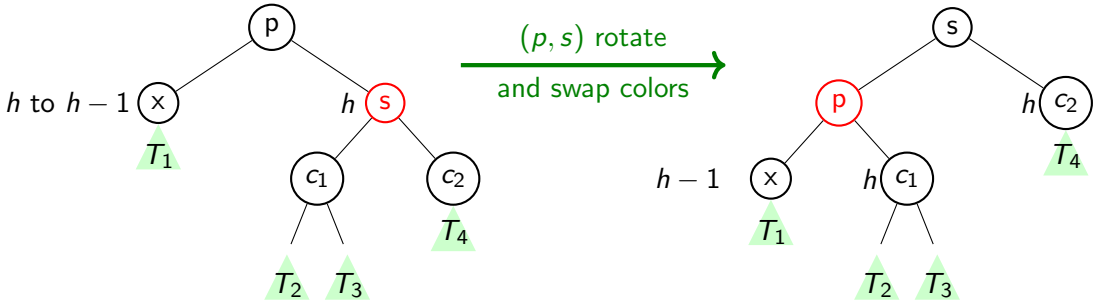


Violation solved!

Case 2: x is not **red** and root.

Do nothing.

Case 3: x is not **red** and the sibling of x is **red**



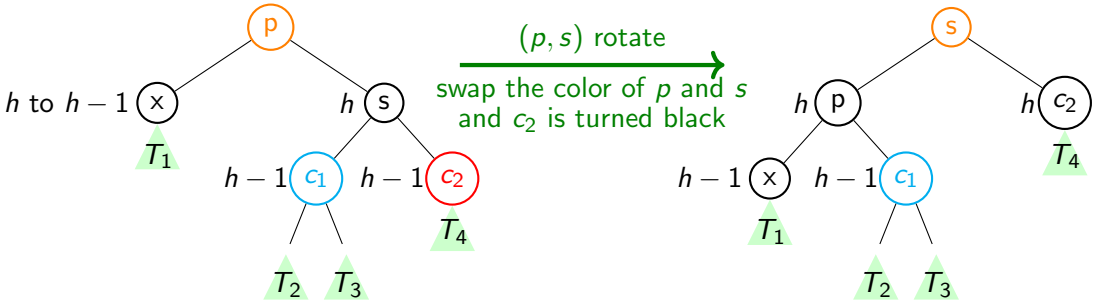
The transformation does not solve the height violation at the parent of x but changes the sibling of x from **red** to **black**.

Exercise 7.8

Why must p, c_1, and c_2 be non-null and **black***?*

Case 4.1: x is not **red**, the sibling of x is **black**, and the right nephew is **red** (Assuming x is the left child)

The color of p and c_1 does not matter.

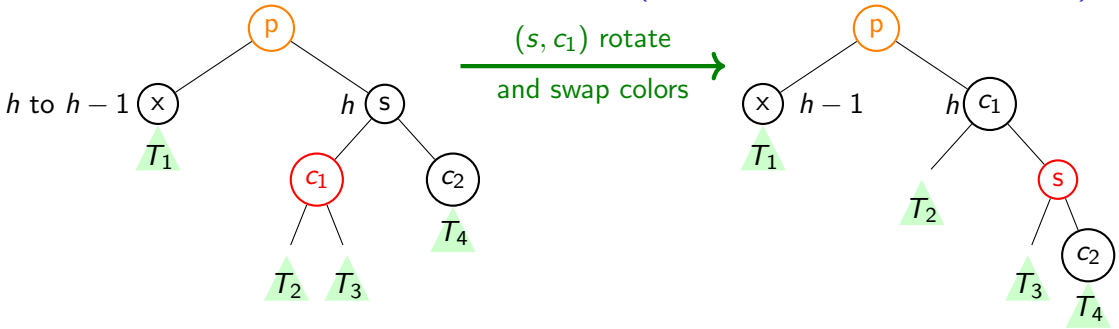


The above transformation solves the black height violation.

Exercise 7.9

Write the above case if x is the right child of p.

Case 4.2: x is not **red** and the sibling is **black**, and the left and right nephews are **red** and not **red** respectively (Assuming x is the left child)



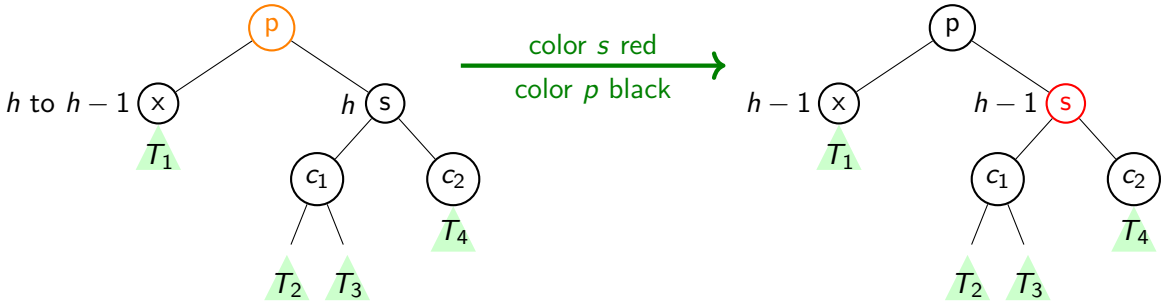
The above transformation does not solve the height violation. It changes the right child of the sibling from not **red** to **red**, which is the case 4.1.

Exercise 7.10

a. Why can case 4.1 transformation not be applied to case 4.2?

b. Write the above case if x is the right child of p.

Case 4.3: x is not **red**, the sibling of x is **black**, and the nephews of x are not **red**



The above transformation reduces $bh(p)$ by 1, if p was black, and there may be a potential violation at p, which is at the lower level.

We apply the case analysis again at node p. The only case that kicks the can upstairs!! All cases are covered.

Structure among cases

- Cases 1, 2, and 4.1 solve the violation at the node.
- Case 3 turns the violation into 4.1 or 4.2.
- Case 4.2 turns the violation into 4.1.
- Case 4.3 moves the violation from x to its parent p.

Summary of deletion

1. Delete like BST. There may be a black height violation at the child of the deleted node.
2. While case 4.3 occurs, re-color nodes and move up the black height violation.
3. For all the other cases, we rotate or re-color, and the violation is finished.

Topic 7.5

Tutorial problems

Exercise: validity of rotation

Exercise 7.11

Prove that after rotation the resulting tree is a binary search tree.

Exercise: sorted insert

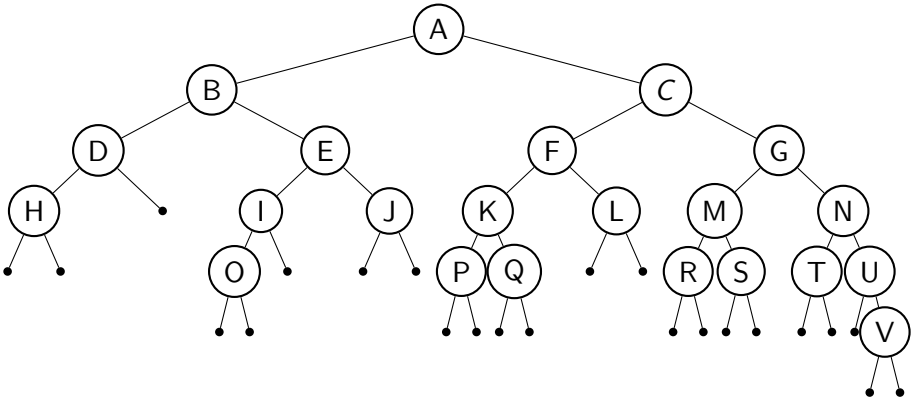
Exercise 7.12

Insert sorted numbers 1,2,3,..., 10 into an empty red-black tree. Show all intermediate red-black trees.

Insert and delete

Exercise 7.13

Consider the tree below. Can it be colored and turned into a red-black tree? If we wish to store the set 1, . . . , 22, label each node with the correct number. Now add 23 to the set and then delete 1. Also, do the same in the reverse order. Are the answers the same? When will the answers be the same?



Exercise: Hash table vs Red-black tree

Exercise 7.14

a. Give running time complexities of delete, insert, and search in the red-black tree.

b. What are the advantages of the red-black tree compared to the Hash table, where every operation (search, insert, delete) is almost constant time?

Topic 7.6

Extra slides: AVL trees (In GATE/GRE syllabus)

AVL (Adelson, Velsky, and Landis) tree

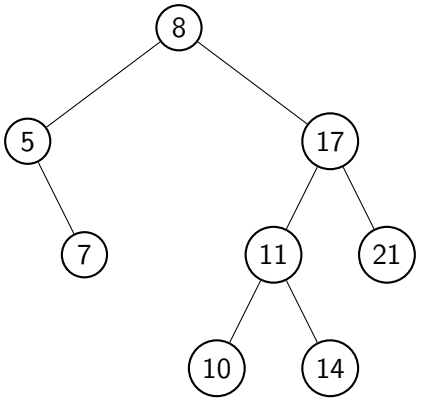
Definition 7.3

An AVL tree is a binary search tree such that for each node n

$$|height(right(n)) - height(left(n))| \leq 1.$$

Example 7.8

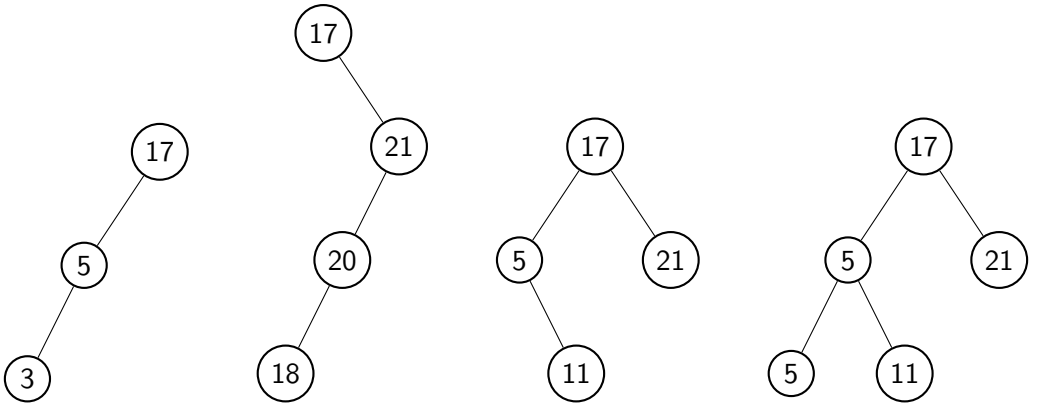
An example of an AVL tree.



Exercise: Identify the AVL trees

Exercise 7.15

Which of the following are AVL trees?



Topic 7.7

Height of AVL tree

AVL tree height

Theorem 7.1

The height of an AVL tree T having n nodes is $O(\log n)$.

Proof.

Let $n(h)$ be the minimum number of nodes for height h.

Base case:

$n(1) = 1$ and $n(2) = 2$.

Induction step:

Consider an AVL tree with height $h \geq 3$. In the minimum case, one child will have a height of $h-1$ and the other child will have a height of $h-2$. (Why?)

Therefore, $n(h) = 1 + n(h-1) + n(h-2)$. ...

Commentary: We need to show that $n(h) > n(h-1)$ is monotonous. Ideally, $n(h) = 1 + n(h-1) + min(n(h-2), n(h-1))$. This proves that $n(h) > n(h-1)$.

AVL tree height(2)

Proof(continued.)

Since $n(h-1) > n(h-2)$,

$$n(h) > 2n(h-2).$$

Therefore,

$$n(h) > 2^i n(h-2i).$$

For $i = h/2 - 1$(Why?),

$$n(h) > 2^{h/2-1} n(2) = 2^{h/2}.$$

Commentary: Here is the explanation of the last step. Consider an AVL tree with m nodes and h height. By definition, $h(n) \leq m$. Since $h < 2\log n(h)$, $h < 2\log m$. Therefore, h is $O(\log m)$.

Therefore,

$$h < 2\log n(h).$$

Therefore, the height of an AVL tree is $O(\log n)$. (Why?) □

Closest leaf

Theorem 7.2

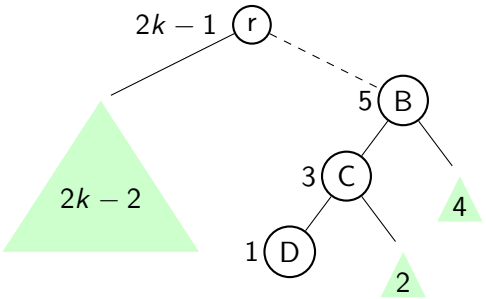
Let T be an AVL tree. Let the level of the closest leaf to the root of T is k.

$$height(T) \leq 2k - 1$$

Proof.

Let D be the closest leaf of the tree.

- The height of $right(C)$ cannot be more than 2. (Why?)
- Therefore, the maximum height of C is 3.
- Therefore, the maximum height of $right(B)$ is 4.
- Therefore, the maximum height of B is 5.
- Continuing the argument, the maximum height of root r is $2k - 1$.



□

A part of AVL is a complete tree

Theorem 7.3
Let T be an AVL tree. Let the level of the closest leaf to the root of T is k. Upto level $k-2$ all nodes have two children.

Proof.
A node at level $k-2-i$ cannot be a leaf. (Why?)

Let us assume that a node n at level $k-2-i$ has a single child n'.

The height of n' cannot be more than 1. (Why?)

Therefore, n' is a leaf. Contradiction. □

Exercise 7.16
Show T has at least 2^{k-1} nodes.

Another proof of tree height bound

Let T have n nodes and the height of T be h.

We know the following from the previous theorems.

- $n \geq 2^{k-1}$, and
- $2k - 1 \geq h$.

Therefore,

$$n \geq 2^{k-1} \geq 2^{(h-1)/2}$$

Exercise 7.17

What is the maximum number of nodes given height h?

Problem: A sharper bound for the AVL tree

Exercise 7.18

a. Find largest c such that $c^{k-2} + c^{k-1} \geq c^k$

b. Recall $n(h) = 1 + n(h-1) + n(h-2)$. Let c_0 be the largest c. Show that $n(h) \geq c^{h-1}$.

c. Prove that the above bound is a sharper bound than our earlier proof.

Topic 7.8

Insertion and deletion in AVL trees

Insert and delete

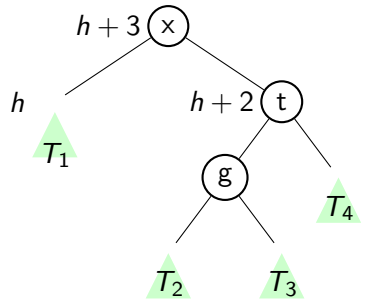
Insert and delete like BST.

At most a path to one node may have height imbalances of 2. (Why?)

We have to repair height imbalances by rotations around the deepest imbalanced node.

Rebalancing AVL trees

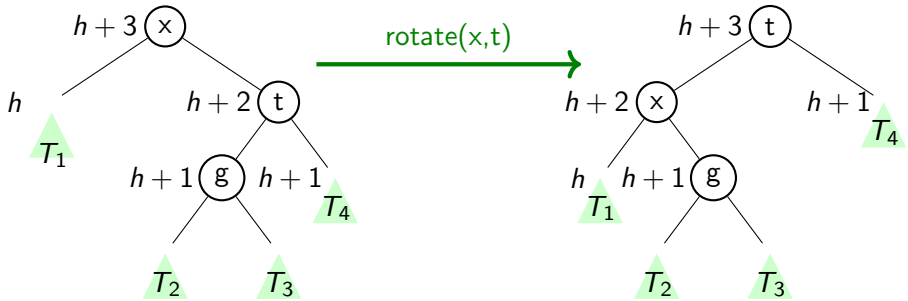
Let x be the deepest imbalanced node. Let t be the taller child. Let g be the grandchild via t that is not on the straight path from x.



Three cases:

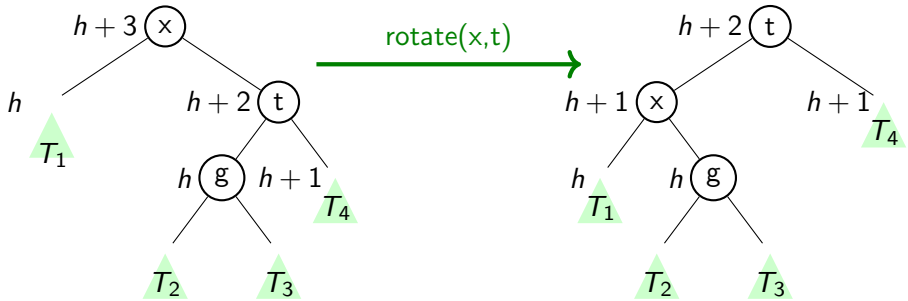
1. Case 1: Height of g is $h+1$ and T_4 is $h+1$.
2. Case 2: Height of g is h and T_4 is $h+1$.
3. Case 3: Height of g is $h+1$ and T_4 is h.

Case 1: Both grandchildren via t have height $h+1$



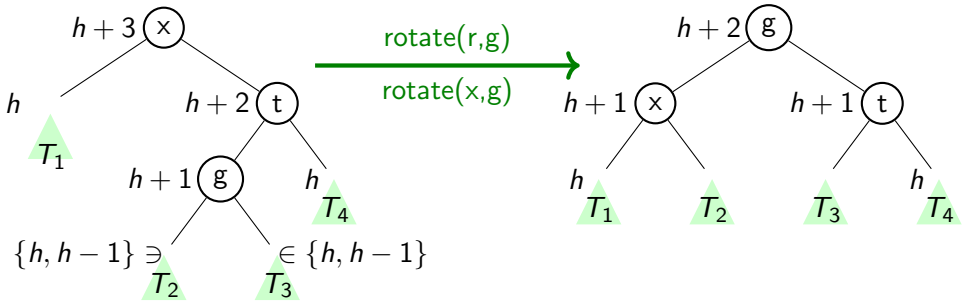
The imbalance in the subtree is repaired. We check the parent of t.

Case 2: Right-left grandchild has height h



Imbalance is repaired. But, the parent of t may need repair.

Case 3: Right right grandchild has height h



Imbalance is repaired. But, the parent may need repair.

Complexity of insertion/deletion

Exercise 7.19

a. What is the bound on the number of rotations for a single insert/delete?

b. Compare the bounds with RB trees insertion/deletion.

c. Which definition is more strict RB or AVL? Or, are they incomparable?

End of Lecture 7

CS213/293 Data Structure and Algorithms 2024

Lecture 8: Heap

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-09-02

Topic 8.1

Priority queue

Scheduling problem

On a computational server, users are submitting jobs to run on a single CPU.

- A user also declares the expected run time of the job.
- Jobs can be preempted.

Policy: shortest remaining processing time, which allows interruption of a job if a new job with a smaller run time is submitted.

The policy minimizes average waiting time.

Scheduling problem operations

We need the following operations in the scheduling problem.

- Update the remaining time in every tick
- Delete a job when the remaining time is zero
- Find the next job to run
- Insert a job when it arrives

Definition 8.1

In a priority queue, we dequeue the highest priority element from the enqueue elements with priorities.

Interface of priority queue

https://en.cppreference.com/w/cpp/container/priority_queue

- `priority_queue<T,Container,Compare> q` : allocates new queue q
- `q.push(e)` : adds the given element e to the queue.
- `q.pop()` : removes the highest priority element from the queue.
- `q.top()` : access the highest priority element.

- `Container` class defines the physical data structure where the queue will be stored. The default value is `Vector`.
- `Compare` class defines the method of comparing priorities of two elements.

Topic 8.2

Implementations of priority queue

Implementation using unsorted linked list/array

In case we use a linked list,

- We implement `q.push` by inserting the element at the front of the linked list, which is $O(1)$ operation.
- We need to scan the entire list to find the maximum for implementing `q.pop` and `q.top`

Exercise 8.1

How will we implement a priority queue over unsorted arrays?

Implementation using sorted linked list/array

In case we use a linked list,

- The maximum will be at the end of the list. We can implement `q.pop` and `q.top` in $O(1)$.
- However, `q.push(e)` needs to scan the entire list to find the right place to insert e, which is $O(n)$ operation.

Priority queue

The priority queue is one of the fundamental containers.

Many other algorithms assume access to efficient priority queues.

We will define a data structure heap that provides an efficient implementation for the priority queue.

Commentary: The heap is like the red-black tree, which provides an efficient implementation for ordered maps.

Topic 8.3

Heap - somewhat sorting!

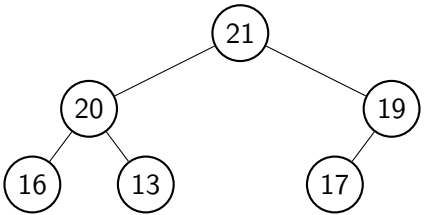
Heap

Definition 8.2

A heap T is a binary tree such that the following holds.

- *(structural property) All levels are full except the last one and the last level is left filled.*
- *(heap property) for each non-root node n, $key(n) \leq key(parent(n))$.*

Example 8.1



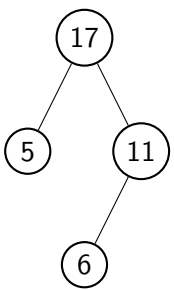
Exercise 8.2

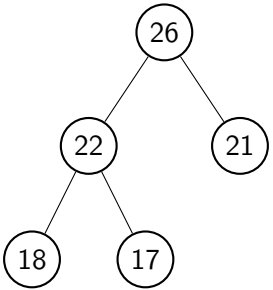
a. Show that nodes on a path from the root to a leaf have keys in non-increasing order.
b. The above definition is called maxheap. Can we symmetrically define minheap?

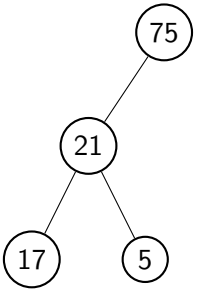
Exercise: Identify Heap

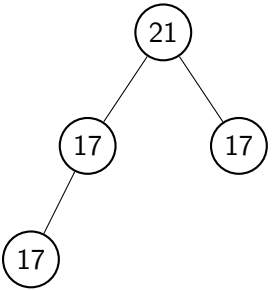
Exercise 8.3

Which of the following are Heaps?









Algorithm: maximum

Algorithm 8.1: MAXIMUM(Heap T)

return $T[0]$

- Correctness
  - Let us suppose the maximum is not at the root.
  - There is a node n that has maximum key but $parent(n)$ has greater key, which violates heap condition.
  - Contradiction.
- Running time is $O(1)$.

Height of heap

Let us suppose a heap has n nodes and height h.

The number of nodes in a complete binary tree of height h is $2^h - 1$.

Therefore,

$$2^{h-1} - 1 < n \leq 2^h - 1.$$

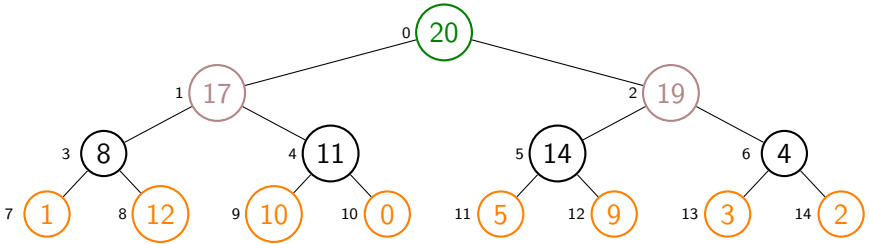
Therefore $n = \lfloor \log_2 n \rfloor$

Exercise 8.4

Give an example of a heap that touches the lower bound.

Storing heap

Let us number the nodes of a heap in the order of level.



$parent(i) = (i-1)/2$, $left(i) = 2i+1$, and $right(i) = 2i+2$.

We place the nodes on an array and traverse the heap using the above equations.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
20	17	19	8	11	14	4	1	12	10	0	5	9	3	2

Since the last level is left filled, we are guaranteed the nodes are contiguously placed.

Instead of writing $key(i)$ for node i in heap T, we will write $T[i]$ to indicate the key.

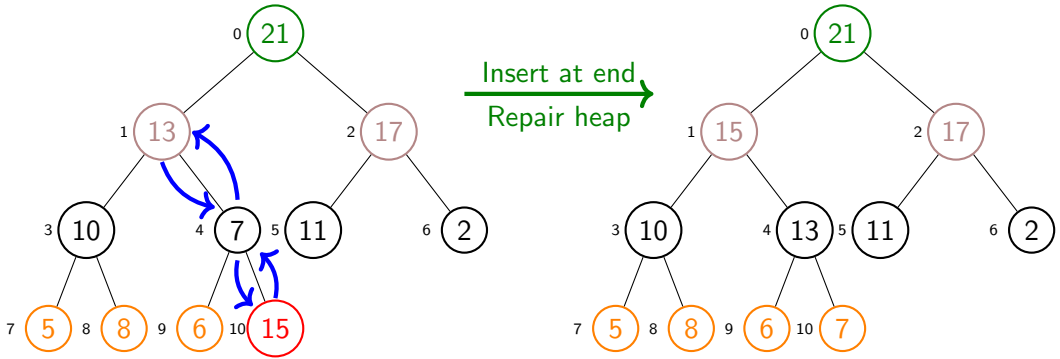
Topic 8.4

Insert in heap

Example: Insert in Heap

Example 8.2

Where do we insert 15?



- Insert at the first available place, which is easy to spot. (Why?)
- Move up the new key if the heap property is violated.

Algorithm: Insert

Algorithm 8.2: INSERT(Heap T, key k)

```
1 i := T.size;
2 T[i] := k;
3 while i > 0 and T[parent(i)] < T[i] do
4 |   SWAP(T, parent(i), i);
5 |   i := parent(i)
6 T.size := T.size + 1;
```

- Correctness
  - Structural property holds due to the insertion position.
  - Due to the heap property of input T, the path to i (not including i) the nodes must be in non-increasing order.
  - Let i_0 be the value of i when the loop exits.
  - INSERT replaces the keys of the nodes in the path from i_0 to $T.size$ with the keys of their parents, which implies the keys do not decrease at the internal nodes.
  - Therefore, no introduction of a violation.
  - Therefore, we will have a heap at the end.
- Running time is $O(\log T.size)$.

Exercise 8.5

Why do we need the phrase "not including" and "internal" in the above proof?

Topic 8.5

Heapify: fix the almost heaps

Heapify : a basic operation on a heap

Input to HEAPIFY:

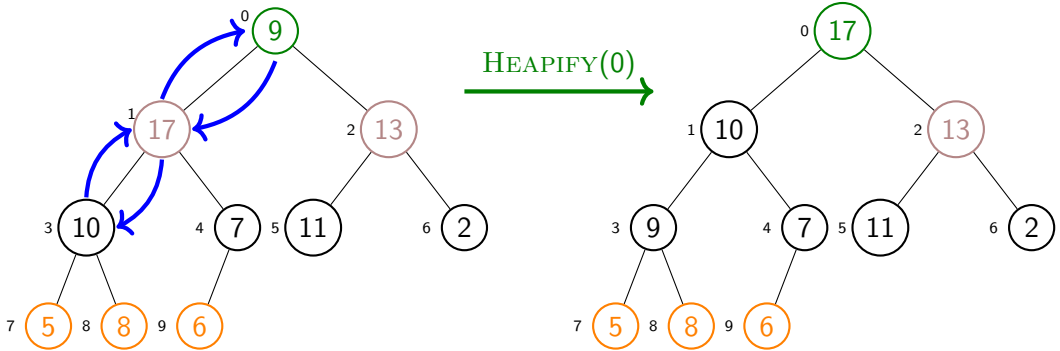
- Let i be a node of a binary tree T with the structural property of heap
- Let us suppose the binary trees rooted at $left(i)$ and $right(i)$ are valid heaps.
- $T[i]$ may be smaller than its children and violates the heap property.

Output of HEAPIFY:
HEAPIFY makes the binary tree rooted at i a heap by pushing down $T[i]$ in the tree.

Example: HEAPIFY

Example 8.3

The trees rooted at positions 1 and 2 are heaps. We have a violation at position 0. *Heapify will fix the problem by moving the key down.*



- Keep moving down to the child which has the maximum key. (Why?)

Algorithm: Heapify

Algorithm 8.3: HEAPIFY(Heap T, i)

$c :=$ INDEXWITHLARGESTKEY$(T, i, left(i), right(i))$ //assume $T[i] = -\infty$ if $i \geq T.size$.
if $c == i$ **then return**;
SWAP(T, c, i);
HEAPIFY(T, c);

- Correctness
  - Same as insert, but we are pushing down.
- Running time is $O(\log T.size)$.

Commentary: Assumption $T[i] = -\infty$ if $i \geq T.size$ is a convenience of notation. We may have a situation, where the $T[i]$ exists and has some key. Without loss of correctness, we can interpret them as if the key is $-\infty$. We will need this interpretation later for HEAPSORT.

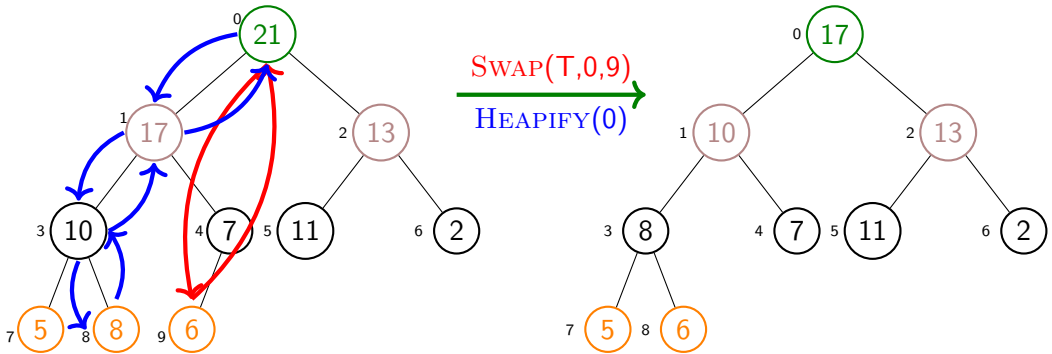
Topic 8.6

Delete maximum in heap

Example: DeleteMax

Example 8.4

Let us delete 21 at position 0.



- Swap with the last position, delete the last position, and run Heapify.

Algorithm: DeleteMax

Algorithm 8.4: DeleteMax(Heap T)

```
1 Swap(T, 0, T.size − 1);
2 T.size := T.size − 1;
3 Heapify(T, 0);
4 return T[T.size];
```

- Correctness
  - The maximum element is removed and heapify returns a heap.
- Running time is $O(\log T.size)$.

Topic 8.7

Build heap

Build heap https://en.cppreference.com/w/cpp/algorithm/make_heap

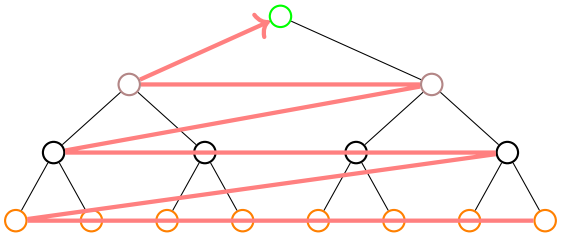
- Input: A binary tree T that has the structural property
  - If the structural property holds, then the T is an array
- Output: A heap over elements of T

Algorithm: BuildHeap

Algorithm 8.5: BUILDHEAP(Heap T)

```
1 for i := T.size − 1 down to 0 do
2     HEAPIFY(T, i)
```

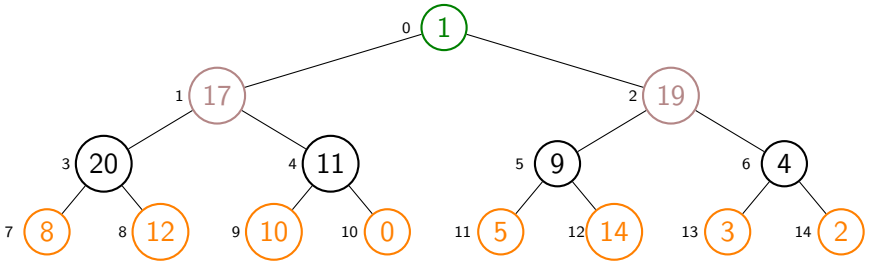
Order of processing in BUILDHEAP.



Example: BuildHeap

Example 8.5

Consider sequence 1 17 19 20 11 9 4 8 12 10 0 5 14 3 2. Let us fill them in the following tree.



BuildHeap traverses the tree bottom up. Heapify calls execute only the following swaps.

- Heapify(T,5): Swap(T,5,12)
- Heapify(T,1): Swap(T,1,3)
- Heapify(T,0): Swap(T,0,1); Swap(T,1,3); Swap(T,3,8);

The other calls to Heapify will not apply any swaps.

Correctness of BuildHeap

- Correctness by induction
  - **Base case:**
    If i does not have children, it is already a heap.
  - **Induction step:**
    We know $left(i) > i$ or $right(i) > i$.
    Due to the induction hypothesis, both the subtrees are heap before processing i.
    Therefore, $\textsc{Heapify}(T, i)$ will return a heap rooted at i.

Running time of BuildHeap

Let us suppose T is a complete tree with n nodes.

Recall: Heapify for a node at height h has $O(h)$ swaps.

At height h the number of nodes is $\lceil n/2^{h+1} \rceil$ and the height of T is $\lfloor \log n \rfloor$.

The total running time of BuildHeap is

$$\sum_{h=0}^{\lfloor \log n \rfloor} O(h)\lceil n/2^{h+1} \rceil = O(\frac{n}{2} \sum^{\lfloor \log n \rfloor} \frac{h}{2^h})$$

Commentary: We used identities $O(f)g = O(fg)$ and $O(f) + O(g) = O(f + g)$.

Since $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$, the running time is $O(n)$.

Calculation to show $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$

We know

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}$$

After differentiating over x,

$$\sum_{h=0}^{\infty} hx^{h-1} = \frac{1}{(1-x)^2}$$

After multiplying with x,

$$\sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2}$$

After putting $x = 1/2$,

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

Topic 8.8

Heapsort

Heapsort

Algorithm 8.6: HEAPSORT(Tree T)

```
1 T.size = |nodes of T|;
2 BUILDHEAP(T);
3 while T.size > 0 do
4     DELETEMAX(T)
```

- Since DELETEMAX moves maximum to $T.size - 1$ position, the array is sorted in place.
- Running time:
  - BUILDHEAP is $O(n)$
  - DELETEMAX(T) is $O(\log i)$ at size i.
- Total running time: $O(n \log n)$.

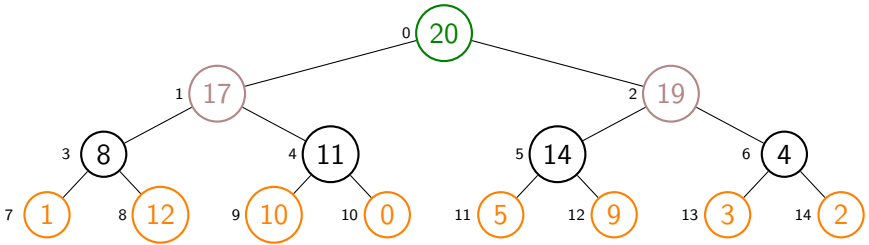
Exercise 8.6

Both BUILDHEAP *and the above loop have iterative runs of* HEAPIFY.
Why are their running time complexities different?

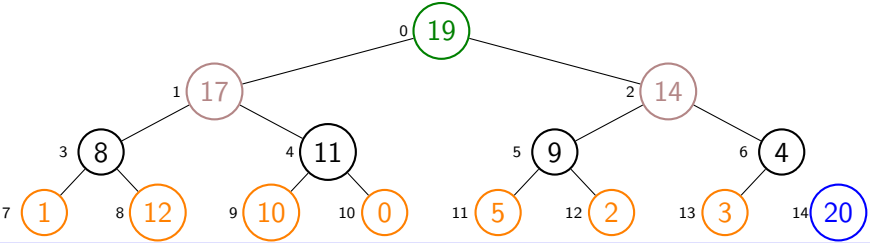
Commentary: Please solve the above exercise to clearly understand the relevant mathematics.

Example: Heapsort

Consider the following Heap obtained after running BuildHeap.

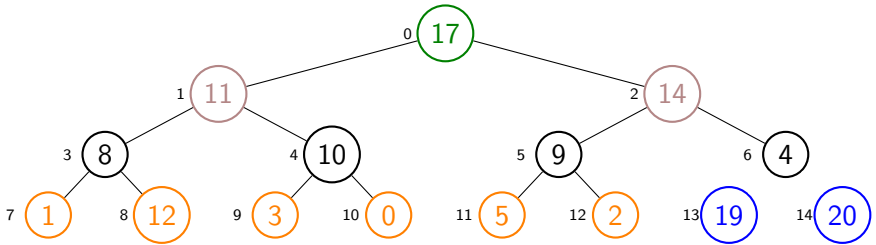


After the first DeleteMax,



Example: Heapsort(2)

After the second DeleteMax,



DeleateMax has placed 19 and 20 at their sorted position.

Topic 8.9

Tutorial problems

Exercise: implement scheduling problem

Exercise 8.7

Give an implementation for the scheduling problem using Heap.

Exercise: Why heap?

Exercise 8.8

Can a Priority Queue be implemented as a red-black tree? What advantages does a heap implementation have over a red-black tree implementation?

Exercise: BST and Heap (Midterm 2023)

Exercise 8.9

Give a tree, if exists, that is a binary search tree, is a heap, and has more than two nodes. If such a tree does not exist, give a reason.

Exercise: 2D-matrix

Exercise 8.10

Suppose we have a 2D array where we maintain the following conditions: for every (i,j), we have $A(i,j) \leq A(i+1,j)$ *and* $A(i,j) \leq A(i,j+1)$. *Can this be used to implement a priority queue?*

Exercise: kth smallest element

Exercise 8.11

Given an unsorted array find the kth smallest element using a priority queue.

Exercise: Merge heaps (Midterm 2023)

Exercise 8.12

Given two heaps give an efficient algorithm to merge the heaps.

End of Lecture 8

CS213/293 Data Structure and Algorithms 2024

Lecture 9: Pattern matching

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-09-13

Topic 9.1

Pattern matching problem

Pattern matching

Definition 9.1

In a pattern-matching problem, we need to find the position of all occurrences of a pattern string P in a string T.

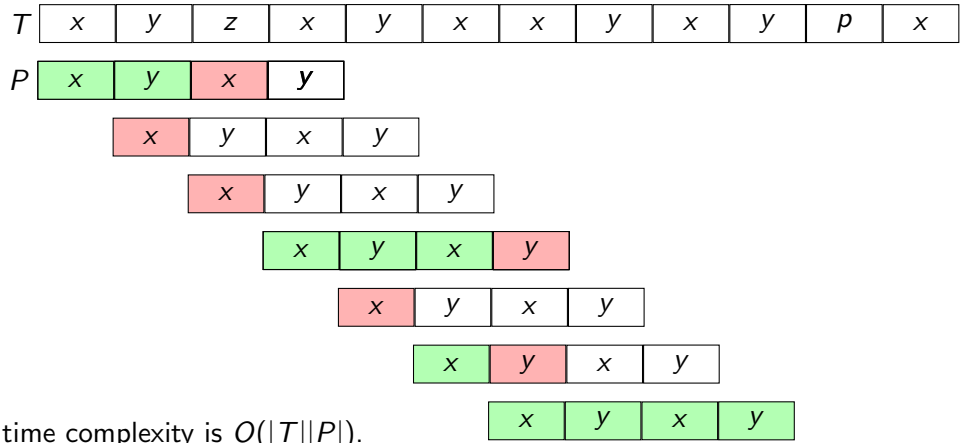
Usage:

- Text editor
- DNA sequencing

Example : Näive approach for pattern matching

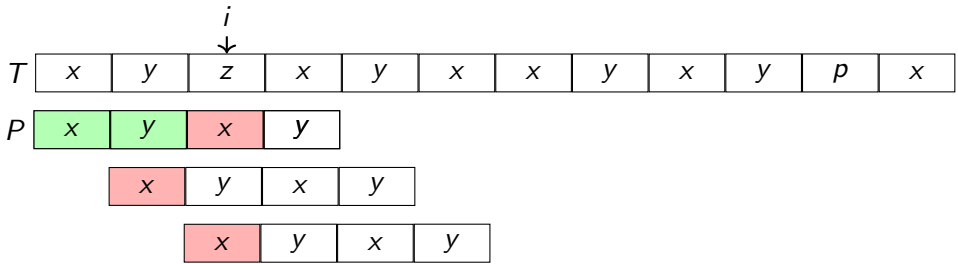
Example 9.1

Consider the following text T and pattern P. We try to match the pattern in every position.



Running time complexity is $O(|T||P|)$.

Wasteful attempts of matching.



Should we have tried to match the pattern at the second and third positions?

No.

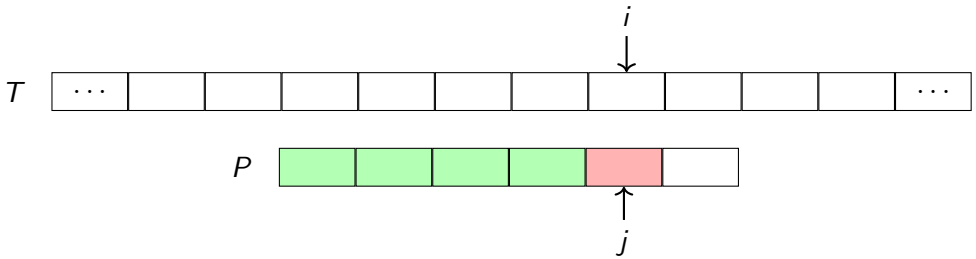
Commentary: In the drawing i is 2. However, we have named the position i to illustrate the argument using symbolic expressions.

Let us suppose we failed to match at position i of T and position 2 of P.

- We know that $T[i-1] = y$. Therefore, there is no matching starting at $i-1$. (Why?)
- We know that $T[i] \neq x$. Therefore, there is no matching starting at i. (Why?)

Shifting the pattern

Let us suppose at position i of T and j of P the matching fails.



Let us suppose we want to resume the search by only updating j.

If we assign j some value k, we are shifting the pattern forward by $j - k$.

Exercise 9.1

What is the meaning of $k = j - 1$, $k = 0$, or $k = -1$?

Side note: out-of-bounds access of P

If k takes value -1 or $|P|$, $P[k]$ is accessing the array out of bounds.

For consistency of the definitions, we will say $P[-1] = P[|P|] = Null$.

However, the algorithms will be carefully written and there will be no out-of-bound access in them.

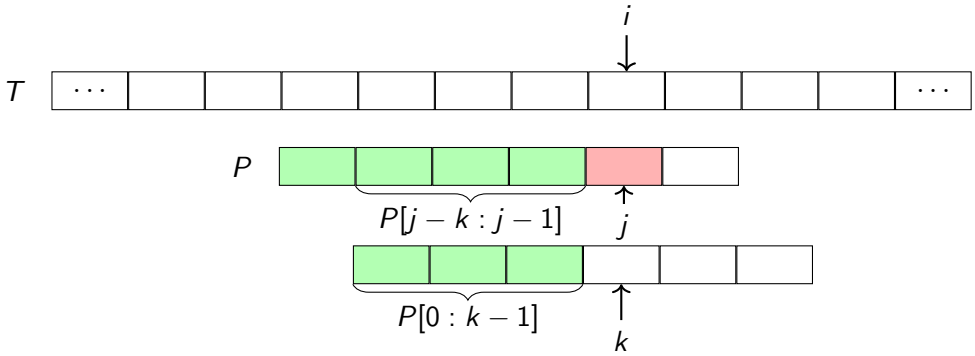
Definition 9.2

Let $P[i:j]$ indicates the array containing elements $P[i], ...P[j]$.

Commentary: In a formal definition, we may overlook or simplify some implementation issues. This allows us to write clean mathematical definitions. However, the implementations need to be careful about the issues.

What is a good value of k?

We know $T[i-j:i-1] = P[0:j-1]$ and $T[i] \neq P[j]$.



We must have $P[0:k-1] = P[j-k:j-1]$ and $P[j] \neq P[k]$(Why?).

Exercise 9.2

Should we choose the largest k or smallest k?

The largest k implies the minimum shift

We choose the largest k such that

$$P[0 : k-1] = P[j-k : j-1] \text{ and } P[j] \neq P[k].$$

k only depends on P and j. Since P is typically small, we pre-compute array h such that $h[j] = k$.

Example 9.2

P	x	y	x	y
h	-1	0	-1	0	2

P	x	y	x	z
h	-1	0	-1	1	0

We can compute h in $O(|P|)$ time. We will discuss this later.

Exercise 9.3

a. Show that $j > h(j) \geq -1$ for each $j \in [0..|P|)$

b. Show that $|P| > h(|P|) \geq 0$ if $|P| > 0$. Is it true if $|P| = 0$?

c. If we drop condition $P[j] \neq P[k]$, what may go wrong?

Commentary: Answer of b: Since $P[|P|] = null$, we are guaranteed that $P[|P|] \neq P[0]$. Since we have $P[0 : -1] = P[j : j-1]$. $k = 0$ will satisfy the condition for $P[|P|]$. Since we are looking the largest k, $P[|P|] \geq 0$.

Knuth–Morris–Pratt algorithm

Algorithm 9.1: KMP(string T,string P)

```
1  assume(|P| > 0);
2  i := 0; j := 0;  found := ∅;
3  h := KMPTable(P);
4  while i < |T| do
5      if P[j] = T[i] then
6          i := i + 1;   j := j + 1;
7          if j = |P| then
8              found.insert(i − j);
9              j := h[j];
10     else
11         j := h[j];
12         if j < 0 then
13             i := i + 1;  j := j + 1;
14 return found
```

Running time complexity:

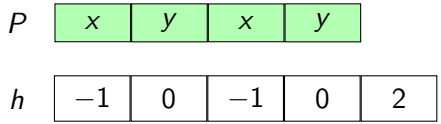
- Since no. of increments of $i \leq |T|$, the line 6 and 13 will execute $\leq |T|$ times in total.
- How do we bound the number of iterations when the **else** branch does not increment i?
  1. The **else** branch reduces j because $h[j] < j$.
  2. Since every time at the loop head $j \geq 0$ (Why?), no. of reductions of $j \leq$ no. of increments of j.
  3. Since i and j are always incremented together, no. of reductions of $j \leq$ no. of increments of i.
  4. no. of reductions of $j \leq |T|$.
- $O(|T|)$ algorithm

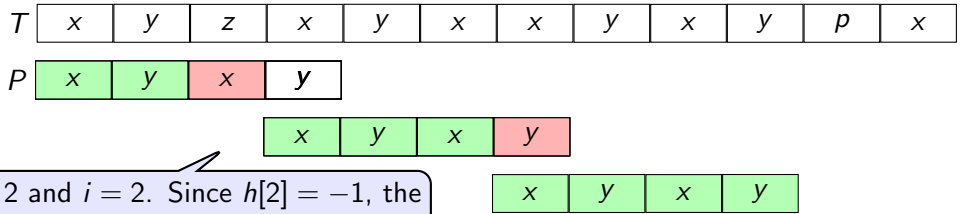
Commentary: The step two is bounding the number of reductions over all iterations of the loop (needs some thinking). It is called amortized complexity. Note that the argument does not guarantee a constant bound over the number of consecutive reduction steps.

Example : KMP execution

Example 9.3

Consider the following text T and pattern P. Let us suppose, we have h.





Shifting at $j = 2$ and $i = 2$. Since $h[2] = -1$, the pattern is shifted to 3, $j = 0$ and $i = 3$.

Topic 9.2

How to compute array h?

Recall: the definition of h

For a pattern P, $h[i]$ is the largest k such that

$$P[0 : k-1] = P[i-k : i-1] \text{ and } P[i] \neq P[k].$$

We use KMP like algorithm again to compute h.

When we compute $h[i]$, we assume we have computed $h[i']$ for each $i' \in [0, i)$.

Self-matching: use KMP again for computing h

We run two indexes i and j on P such that $j < i$.

We assume that for each $k \in (j, i), \neg(P[0 : k-1] = P[i-k : i-1] \wedge P[i] \neq P[k])$.

We will be computing $h[i]$. Let j be the current running match,i.e, $P[i-j : i-1] = P[0 : j-1]$.

When we consider position i, we have two cases.

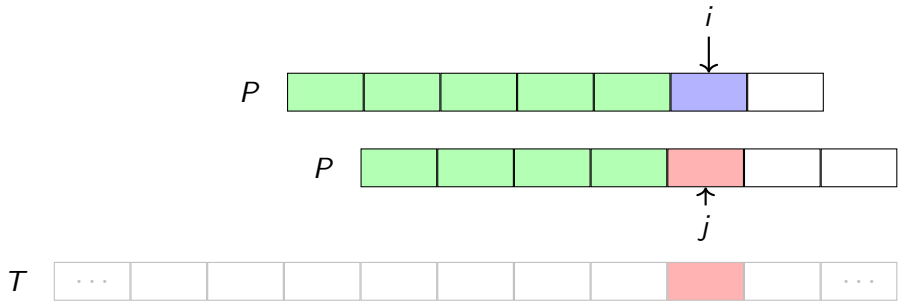
1. $P[i] \neq P[j]$
2. $P[i] = P[j]$

In both the cases, we need to update $h[i]$ and may update j.

We ensure that j is largest by updating j conservatively.

Commentary: Due to the assumption, no k larger j is a valid candidate or $h[i]$.

Case 1: $P[i] \neq P[j]$



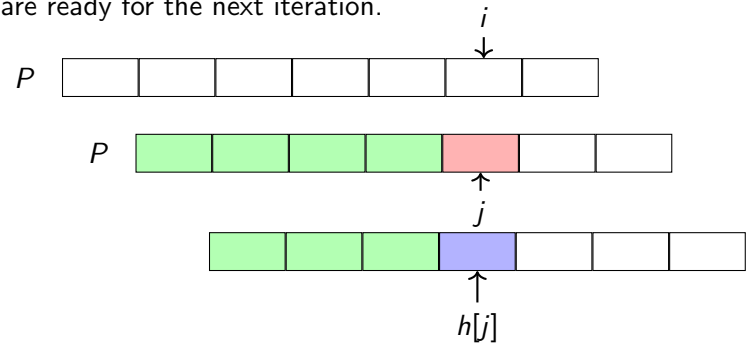
We assign $h[i] := j$, since j meets the requirements.

We have found the shift position for i.Now, we need to prepare for the next index $i + 1$.

Now we need to move the pattern forward as little as possible.

Case 1 (continued): $P[i] \neq P[j]$

After the mismatch, we move the pattern forward as little as possible such that we have a match at position i and are ready for the next iteration.



We must have computed h for earlier indexes. We set $j := h[j]$.

We need to keep reducing j until $P[j] = P[i]$ or $j \leq 0$.

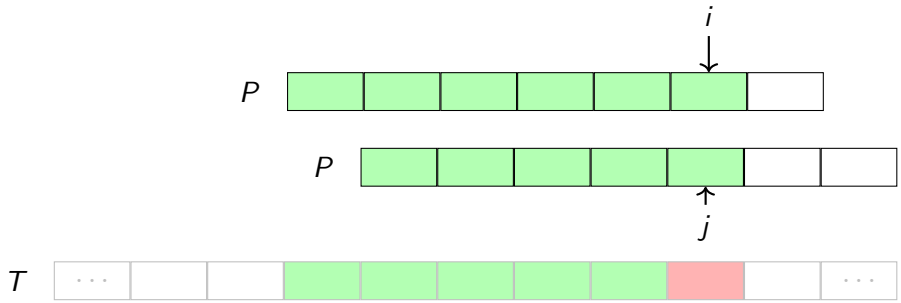
Exercise 9.4

a. Why the value of $h[j]$ be available?

b. Prove that $\forall k \in (h[j], j] : \neg(P[0 : k-1] = P[i-k : i-1] \wedge P[i] \neq P[k])$. *(important point!)*

Case 2: $P[i] = P[j]$

Let us consider the case when matching continues. How should we assign $h[i]$?



We may use $h[i] := j$, but it does not meet the requirement $P[i] \neq P[j]$. (Why?)

Let us jump to $h[j]$, which will meet the requirements. (Why?) We assign $h[i] := h[j]$.

Computing h array

Algorithm 9.2: KMPTable(string P)

$i := 1; j := 0;\ \ h[0] := -1;$
while $i < |P|$ **do**
  if $P[j] \neq P[i]$ **then**
    $h[i] := j;$
    while $j \geq 0$ *and* $P[j] \neq P[i]$ **do**
      $j := h[j];$
  else
    $h[i] := h[j];$
  $i := i + 1; \quad j := j + 1;$
$h[|P|] := j;$
return h

// Prepare for the next iteration

Commentary: Let $prop(i, k) =$ $(P[0 : k-1] = P[i-k : i-1] \wedge P[i] \neq P[k])$. The loop invariant at the head of the outer loop is $P[i-j : i-1] = P[0 : j-1]$, $\forall k \in (j, i), \neg prop(i, k)$, and $\forall l < j\ \ prop(h[l], l) \wedge \forall k \in (h[l], l), \neg prop(l, k)$. We prove the correctness by proving the validity of the loop invariant.

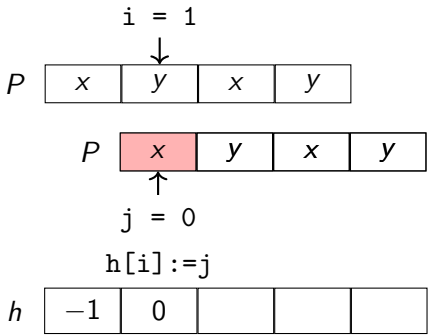
Exercise 9.5

Give proof of correctness of the algorithm.

Example: computing h

Example 9.4

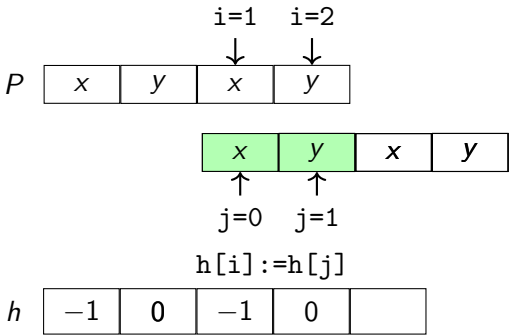
Consider the following pattern P and the first iteration of the outer loop, which is case 1.



We need to update `j := h[j]`. Therefore, `j = -1`.
Afterwards, we increment both j and i. Therefore, `i = 2;j = 0;`.

Example: computing h (cotinued) (2)

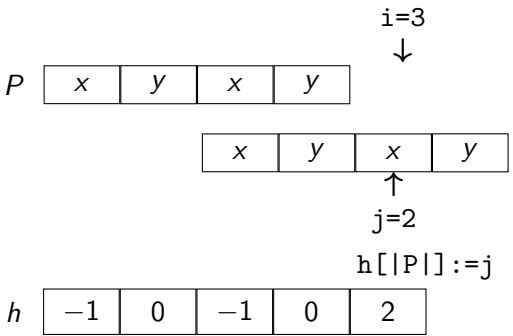
Let us consider the second and third iteration of the outer loop, which are case 2.



After the third iteration, the loop exits since $i \geq |P|$.

Example: computing h (cotinued) (3)

After the third iteration, the loop exists and we update $h[|P|]$.



Topic 9.3

Tutorial problems

Exercise: compute h

Exercise 9.6

Compute array h for pattern "babbaabba".

Exercise: version of KMPTABLE

Exercise 9.7

Is the following version of KMPTABLE *correct?*

Algorithm 9.3: KMPTABLEV2(string P)

```
i := 1; j := 0;  h[0] := −1;
while i < |P| do
    h[i] := j;
    while j ≥ 0 and P[j] ≠ P[i] do
        j := h[j];                    // Moving forward the pattern in minimum steps as in KMP
    i := i + 1;   j := j + 1;
h[|P|] := j;
return h
```

Exercise: compute $h(i)$

Exercise 9.8

Suppose that there is a letter z in P of length n such that it occurs in only one place, say k, which is given in advance. Can we optimize the computation of h?

End of Lecture 9

CS213/293 Data Structure and Algorithms 2024

Lecture 10: Trie: storing *string* $\rightarrow$ *Values*

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-09-09

If keys are strings!

The problem of storing maps boils down to storing keys in an organized manner.

- For unordered keys, we used hash tables
- For ordered keys, we used red-black trees.
- Let us suppose. Our keys are strings.

We have more structure over keys than total order. Can we exploit the structure?

Exercise 10.1

Can we define total order over strings?

Applications of string keys

- Web search
- All occurrences of a text
- Routing table (Keys are IP addresses)

Topic 10.1

Trie

Trie

Let letters of strings be from an alphabet Σ.

Definition 10.1

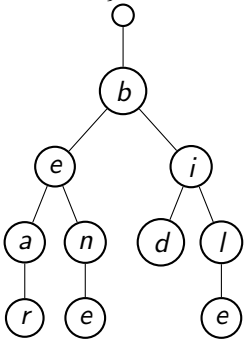
A trie is an ordered tree such that each node except the root is labeled with a letter in Σ and has at most $|\Sigma|$ children.

A trie may store a set of words.

A word stored in a trie is a path from the root to a leaf.

Example 10.1

In the following trie, we store words $\{bear, bile, bid, bent\}$.

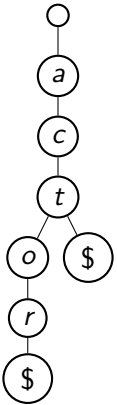


End marker

Sometimes a word is a prefix of another word. We need to add end markers in our trie. In our slides, We will use $.

Example 10.2

Consider set of words $\{act, actor\}$



Running times

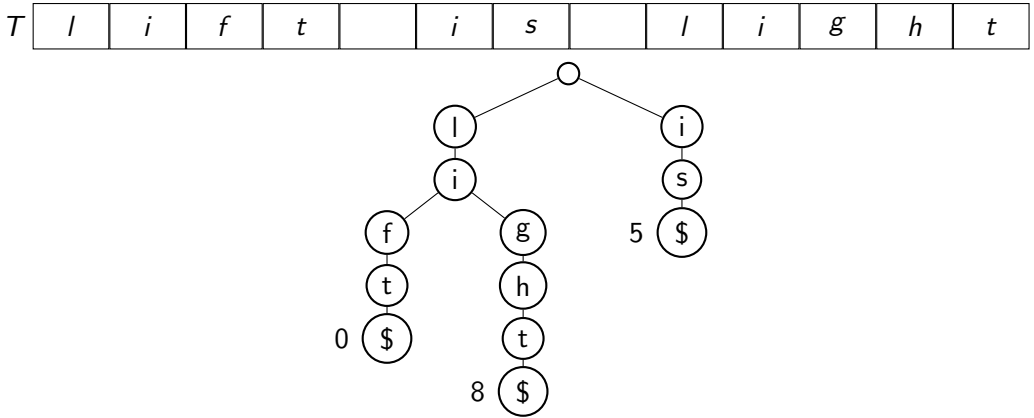
- Storage $O(W)$, W is the total sum of lengths of words
- Find, insert, and delete will take $O(|\Sigma|m)$, where m is the length of the input word
  - At each node, we need to search among children for the node with the next letter.

Application: word search

We may use trie to store the positions of all words of a text. The leaves of trie point at

- the first occurrence position of the word or
- a list of all occurrence positions.

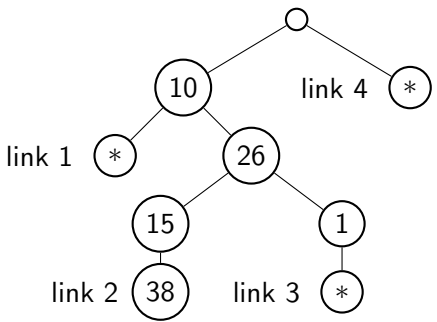
Example 10.3



Application: routing table

Example 10.4

An internet router contains a routing table that maps IP addresses to links attached to the router.



The IP address of a packet is matched with the trie. The link with the longest match will receive the packet.

Exercise 10.2

Which link will receive the packets for following IP address?

- 21.10.1.6
- 10.26.10.6
- 10.26.1.6
- 10.26.15.9

Topic 10.2

Compressed trie

Compressed trie

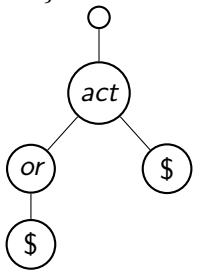
Definition 10.2

A compressed trie is a trie with nodes that do not have single children and nodes are labeled with substrings of words.

Obtained from standard trie by compressing chain of redundant nodes.

Example 10.5

In the following compressed trie, we store words $\{actor, act\}$.



The number of nodes in the compressed trie

Commentary: We did not prove exactly the following theorem. But, we can easily modify our earlier proof.

Theorem 10.1 (Recall)

If each internal node has at least two children, then the number of internal nodes is less than the number of leaves.

Each leaf represents a word.

Therefore, the number of internal nodes in the compressed trie is bounded by the number of words stored in the trie.

Does compression save the space?

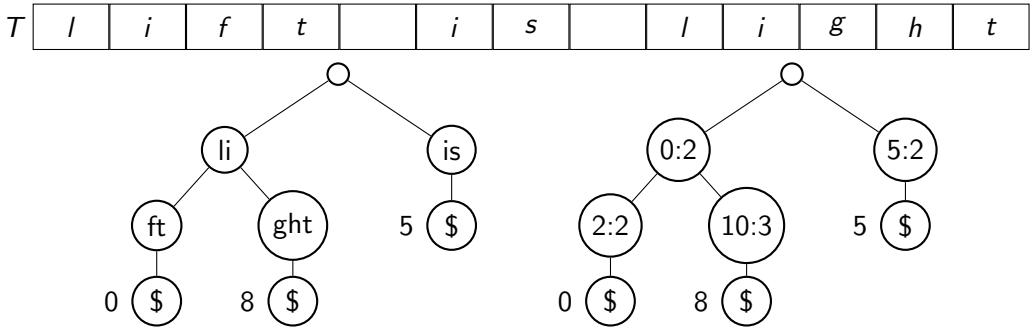
Typical usage of trie: fast search of words on a large text

The text must have been stored separately from the trie.

We need not store the strings on the nodes.

All we need to point at the position of the stored text and the length of the substring.

Example 10.6



Insertion and deletion on trie

Exercise 10.3

Give an algorithm for insertion and deletion in a compressed trie.

Topic 10.3

Suffix tree

Pattern search problem: Another perspective

Typical setting: We search in a (mostly) stable text T using many patterns several times.

Example 10.7

- *A text editor, where text changes slowly and searches are performed regularly.*
- *Searching in well-known large sequences like genomes.*

Can we construct a data structure from the text that allows fast search?

Suffix tree

Definition 10.3

A suffix tree of a text T is a trie built for suffixes of T.

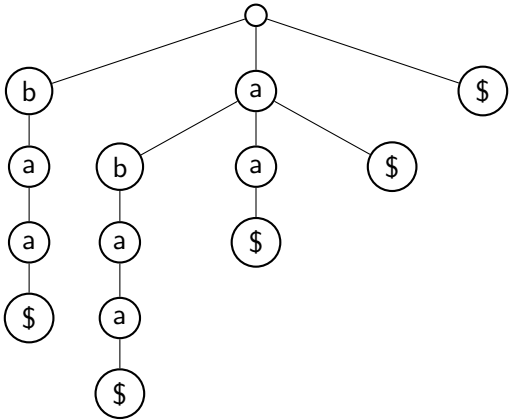
Exercise 10.4

How many leaves are possible for a suffix tree?

Example: suffix tree

Example 10.8

The following is the suffix tree of "abaa".



Usage of suffix tree

- Check if pattern P occurs in T.
- Check if P is a suffix of T.
- Count the number of occurrences of P in T.

Exercise 10.5

Using suffix tree find the longest string that repeats.

Suffix links

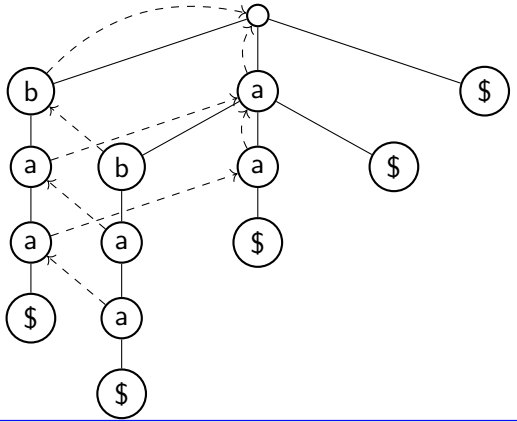
We can solve more interesting problems if we add more structure to our suffix tree.

Definition 10.4

In each node $x\alpha$, we add a pointer suffix link that points to node α.

Example 10.9

The following is the suffix tree of "abaa" with suffix links.



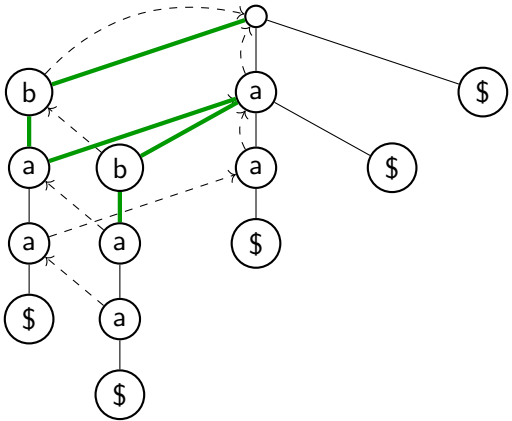
Usage of suffix links

Find the longest sub-string of T and P.

- Walk down the suffix tree following P.
- At a dead end, save the current depth and follow the suffix link from the current node.
- After exhausting P, return the longest substring found.

Example 10.10

The following is the suffix tree of $T =$ "abaa". Let us find the longest sub-string of "baba" and T.



Topic 10.4

Constructing suffix tree

Suffix tree construction

If we have the suffix tree for $T[0:i-1]$, we construct the suffix tree for $T[0:i]$.

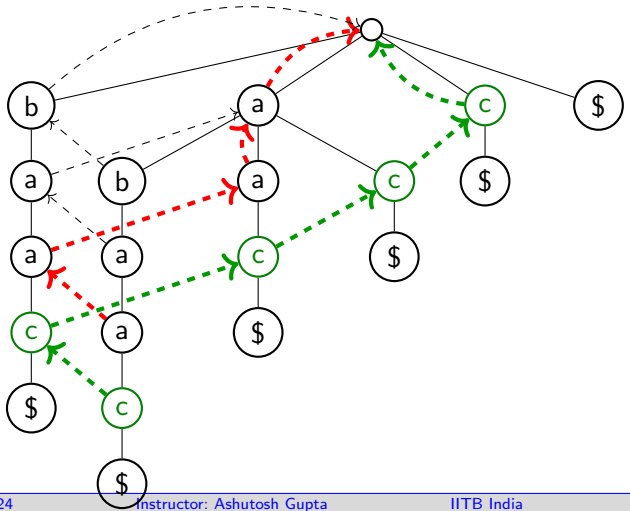
In the order of the suffix links, Insert $T[i]$ at the end of each path of the tree,

Exercise 10.6

a. What is the complexity of the above algorithm?
b. What will be the complexity of suffix tree construction without suffix links?

Example 10.11

Let us add c in the suffix tree of "abaa".



Ukkonen's algorithm

Suffix links give us $O(n^2)$ construction, can we do better?

Yes.

Ukkonen's algorithm uses more programming tricks to achieve $O(n)$. We will not cover the algorithm in this course.

Topic 10.5

Tutorial problems

Exercise: suffix tree

Exercise 10.7

Compute the suffix tree for abracadabra$. Compress degree 1 nodes. Use substrings as edge labels. Put a square around nodes where a word ends. Use it to locate the occurrences of abr.

Exercise: worst-case suffix tree

Exercise 10.8

Review the argument that for a given text T, consisting of k words, the ordinary trie occupies space which is a constant multiple of $|T|$. How is it that the suffix tree for a text T is of size $O(|T|^2)$? Give a worst-case example.

Commentary: Milind notes

End of Lecture 10