Algoritmos

Estructuras de Datos

Grupo de Programación

Campamento de Programación

Contenido

Estructuras de Datos
- Estructuras de Datos Elementales
- Estructuras de Datos Avanzadas

Contenido

Estructuras de Datos
- Estructuras de Datos Elementales
- Estructuras de Datos Avanzadas

Estructuras de Datos

- Una estructura de datos es una forma de organizar data para facilitar su posterior uso

Estructuras de Datos

- Una estructura de datos es una forma de organizar data para facilitar su posterior uso
- Toda estructura de datos define operaciones como inserción, remoción, búsqueda, etc

Estructuras de Datos

- Una estructura de datos es una forma de organizar data para facilitar su posterior uso
- Toda estructura de datos define operaciones como inserción, remoción, búsqueda, etc
- Las estructuras de datos más básicas son los arreglos y las matrices (arreglos bidimensionales)

Estructuras de Datos

- Una estructura de datos es una forma de organizar data para facilitar su posterior uso
- Toda estructura de datos define operaciones como inserción, remoción, búsqueda, etc
- Las estructuras de datos más básicas son los arreglos y las matrices (arreglos bidimensionales)
- El objetivo es que sean un soporte para el diseño de algoritmos eficientes

Estructuras de Datos

- Una estructura de datos es una forma de organizar data para facilitar su posterior uso
- Toda estructura de datos define operaciones como inserción, remoción, búsqueda, etc
- Las estructuras de datos más básicas son los arreglos y las matrices (arreglos bidimensionales)
- El objetivo es que sean un soporte para el diseño de algoritmos eficientes
- C++ provee los llamados *contenedores* que implementan varias estructuras de datos, hay una buena cantidad de estructuras de datos implementadas, las más especializadas pueden ser construidas tomando estas de base.

Contenido

Estructuras de Datos
- Estructuras de Datos Elementales
- Estructuras de Datos Avanzadas

Vectores

- Consideramos como `vectores` a arreglos con memoria dinámica

Vectores

- Consideramos como `vectores` a arreglos con memoria dinámica
- Memoria estática: se conoce el tamaño del arreglo en tiempo de compilación (al momento de compilar el programa)

Vectores

- Consideramos como `vectores` a arreglos con memoria dinámica
- Memoria estática: se conoce el tamaño del arreglo en tiempo de compilación (al momento de compilar el programa)
- Memoria dinámica: **no** se conoce el tamaño del arreglo en tiempo de compilación, sino solo en tiempo de ejecución (al momento de correr el programa)

Vectores

```
#include<iostream>
#include<vector>

using namespace std;

int main(){
  vector<int> V; // no hay necesidad de indicar el tamano inicial
  V.push_back(2);
  V.push_back(1);
  V.push_back(4);
  for ( int i=0; i<V.size(); i++ ) {
    cout << V[i] << endl;
  }
}
```

Pilas

- Es un contenedor que almacena un conjunto de elementos, siendo una memoria tipo LIFO (Last-In First-Out)

Pilas

- Es un contenedor que almacena un conjunto de elementos, siendo una memoria tipo LIFO (Last-In First-Out)
- Solo permite acceso al último elemento agregado

Pilas

- Es un contenedor que almacena un conjunto de elementos, siendo una memoria tipo LIFO (Last-In First-Out)
- Solo permite acceso al último elemento agregado
- Se puede pensar como una forma de apilar platos, uno encima de otro, de esta manera solo se puede retirar el último que fue agregado

Pilas

```
#include<stack>
#include<iostream>

using namespace std;

int main() {
  stack<int> S;
  S.push(3);
  S.push(5);
  S.push(20);
  while ( ! S.empty() ) {
    cout << S.top() << endl;
    S.pop();
  }
}
```

Colas

- Es un contenedor que almacena un conjunto de elementos, siendo una memoria tipo FIFO (First-In First-Out)

Colas

- Es un contenedor que almacena un conjunto de elementos, siendo una memoria tipo FIFO (First-In First-Out)
- Solo permite acceso al primer elemento agregado

Colas

- Es un contenedor que almacena un conjunto de elementos, siendo una memoria tipo FIFO (First-In First-Out)
- Solo permite acceso al primer elemento agregado
- Se puede pensar como una cola del banco, donde cada usuario que llega tiene que esperar que los que llegaron antes son atentidos primero salir de la cola

Colas

```
#include<queue>
#include<iostream>

using namespace std;

int main() {
  queue<int> Q;
  Q.push(5);
  Q.push(2);
  Q.push(9);
  while ( !Q.empty() ) {
    cout << Q.front() << endl;
    Q.pop();
  }
}
```

Colas de prioridad

- Es un contenedor que almacena un conjunto de elementos cada uno con cierta prioridad, siendo una memoria tipo FIFO (First-In First-Out)

Colas de prioridad

- Es un contenedor que almacena un conjunto de elementos cada uno con cierta prioridad, siendo una memoria tipo FIFO (First-In First-Out)
- Solo permite acceso al elemento con mayor prioridad

Colas de prioridad

- Es un contenedor que almacena un conjunto de elementos cada uno con cierta prioridad, siendo una memoria tipo FIFO (First-In First-Out)
- Solo permite acceso al elemento con mayor prioridad
- Se puede pensar como una cola del banco con diferentes tipos de usuarios (clientes regulares, clientes vip, no clientes, ec), donde cada usuario es atendido según su prioridad

Colas de prioridad

```
#include<queue>
#include<iostream>

using namespace std;

int main() {
   priority_queue<int> Q; // la prioridad esta definida por el mism
   Q.push(6);
   Q.push(2);
   Q.push(9);
   while ( ! Q.empty() ){
      cout << Q.top() << endl;
      Q.pop();
   }
}
```

Sets

- Sets o conjuntos representan un conjunto de elementos sin repetición

Sets

- Sets o conjuntos representan un conjunto de elementos sin repetición
- Adicionalmente, los sets preservan el orden de los elementos

Sets

- Sets o conjuntos representan un conjunto de elementos sin repetición
- Adicionalmente, los sets preservan el orden de los elementos
- Mantener el orden implica más esfuerzo computacional, por tanto se pierde un poco de eficiencia

Sets

- Sets o conjuntos representan un conjunto de elementos sin repetición
- Adicionalmente, los sets preservan el orden de los elementos
- Mantener el orden implica más esfuerzo computacional, por tanto se pierde un poco de eficiencia
- Si no se desea mantener el orden y solo se requiere unicidad de los elementos se puede usar `unordered_set`

Sets

```cpp
#include<set>
#include<iostream>

using namespace std;

int main(){
  set<int> S;
  S.insert(7);
  S.insert(3);
  S.insert(7);
  for (auto it : S){
    cout << it << endl;
  }
}
```

Listas enlazadas

- Las listas proveen acceso solo al primer elemento y al último, se debe iterar para acceder al resto

Listas enlazadas

- Las listas proveen acceso solo al primer elemento y al último, se debe iterar para acceder al resto
- Es decir, a diferencia de los vectores, no provee acceso aleatorio (Random Access)

Listas enlazadas

- Las listas proveen acceso solo al primer elemento y al último, se debe iterar para acceder al resto
- Es decir, a diferencia de los vectores, no provee acceso aleatorio (Random Access)
- Para acceder a los elementos hay que iterar la lista

Listas enlazadas

```
#include<list>
#include<iostream>

using namespace std;

int main() {
  list<int> L;
  auto it = L.begin();
  L.insert(it, 4);
  L.insert(it, 2);
  L.insert(it, 9);
  for ( auto itr : L ){
    cout << itr << endl;
  }
}
```

Listas enlazadas vs vectores

- Los vectores son almacenados en espacios de memoria contigua, las listas no

Listas enlazadas vs vectores

- Los vectores son almacenados en espacios de memoria contigua, las listas no
- Por ello, la inserción en una posición cualquiera es lenta para un vector, pero rápida para una lista

Listas enlazadas vs vectores

- Los vectores son almacenados en espacios de memoria contigua, las listas no
- Por ello, la inserción en una posición cualquiera es lenta para un vector, pero rápida para una lista
- Análogamente, la remoción en una posición cualquiera es lenta para un vector, pero rápida para una lista

Listas enlazadas vs vectores

- Los vectores son almacenados en espacios de memoria contigua, las listas no
- Por ello, la inserción en una posición cualquiera es lenta para un vector, pero rápida para una lista
- Análogamente, la remoción en una posición cualquiera es lenta para un vector, pero rápida para una lista
- El uso de uno u otro depende de lo que se necesite

Hash tables

- También llamados diccionarios o maps, son contenedores que asocian dos valores

Hash tables

- También llamados diccionarios o maps, son contenedores que asocian dos valores
- Se pueden pensar como una generalización de los arreglos

Hash tables

- También llamados diccionarios o maps, son contenedores que asocian dos valores
- Se pueden pensar como una generalización de los arreglos
- En un arreglo, cada elemento tiene un índice que es un número entero

Hash tables

- También llamados diccionarios o maps, son contenedores que asocian dos valores
- Se pueden pensar como una generalización de los arreglos
- En un arreglo, cada elemento tiene un índice que es un número entero
- En un map, cada elemento tiene un índice que puede ser cualquier tipo de dato

Hash tables

- También llamados diccionarios o maps, son contenedores que asocian dos valores
- Se pueden pensar como una generalización de los arreglos
- En un arreglo, cada elemento tiene un índice que es un número entero
- En un map, cada elemento tiene un índice que puede ser cualquier tipo de dato
- Se puede pensar como una agenda telefónica, en la que se indexa (o busca) por nombres y apellidos y no por índices enteros

Hash tables

- También llamados diccionarios o maps, son contenedores que asocian dos valores
- Se pueden pensar como una generalización de los arreglos
- En un arreglo, cada elemento tiene un índice que es un número entero
- En un map, cada elemento tiene un índice que puede ser cualquier tipo de dato
- Se puede pensar como una agenda telefónica, en la que se indexa (o busca) por nombres y apellidos y no por índices enteros

Hash tables

```cpp
#include<map>
#include<string>
#include<iostream>

using namespace std;

int main() {
  map<string, int> M;
  M["uno"] = 1;
  M["dos"] = 2;
  M["diez"] = 10;
  for ( auto it : M ){
    cout << it.first << "-" << it.second << endl;
  }
}
```

Hash tables

- De manera análoga a los sets, los maps preservan el orden de los elementos en base al índice

Hash tables

- De manera análoga a los sets, los maps preservan el orden de los elementos en base al índice
- Mantener el orden implica más esfuerzo computacional, por tanto se pierde un poco de eficiencia

Hash tables

- De manera análoga a los sets, los maps preservan el orden de los elementos en base al índice
- Mantener el orden implica más esfuerzo computacional, por tanto se pierde un poco de eficiencia
- Si no se desea mantener el orden se puede usar `unordered_map`

Documentación adicional

- Documentación oficial: https://en.cppreference.com/w/cpp/container
- Geek for geeks: https://www.geeksforgeeks.org/containers-cpp-stl/
- Curso de geek for geeks: https://www.geeksforgeeks.org/courses/cpp-programming-basic-to-advanced

Contenido

Estructuras de Datos
- Estructuras de Datos Elementales
- Estructuras de Datos Avanzadas

Estructuras de Datos Avanzadas

- Para efectos de este curso, entendemos como estructuras de datos avanzadas a aquellas que no son implemendatas por la librería estandar de C++ (STL), sino que deben ser implementadas

Estructuras de Datos Avanzadas

- Para efectos de este curso, entendemos como estructuras de datos avanzadas a aquellas que no son implemendatas por la librería estandar de C++ (STL), sino que deben ser implementadas
- Al implementar una estructura de datos debemos pensar en sus diversas operaciones: inicialización, inserción, remoción, búsqueda, etc. Las operaciones que soporte cada estructura de datos depende de su diseño

Estructuras de Datos Avanzadas

- Para efectos de este curso, entendemos como estructuras de datos avanzadas a aquellas que no son implemendatas por la librería estandar de C++ (STL), sino que deben ser implementadas
- Al implementar una estructura de datos debemos pensar en sus diversas operaciones: inicialización, inserción, remoción, búsqueda, etc. Las operaciones que soporte cada estructura de datos depende de su diseño
- Para tener una estructura de datos eficiente, sus operaciones deben ser eficientes

Estructuras de Datos Avanzadas

- Una situación usual es tener alguna operación computacionalmente costosa (lenta), pero el resto de operaciones eficientes.

Estructuras de Datos Avanzadas

- Una situación usual es tener alguna operación computacionalmente costosa (lenta), pero el resto de operaciones eficientes.
- Si la operación costosa es utilizada pocas veces y las operaciones eficientes muchas veces, en general se tendrá una buena base para un algoritmo

Árboles

- Es una estructura de datos jerárquica, conformado por varios nodos

Árboles

- Es una estructura de datos jerárquica, conformado por varios nodos
- Cada nodo tiene un nodo *padre* y guarda un valor

Árboles

- Es una estructura de datos jerárquica, conformado por varios nodos
- Cada nodo tiene un nodo *padre* y guarda un valor
- El nodo *raíz* es aquel que no tiene ningún padre.

Árboles

- Es una estructura de datos jerárquica, conformado por varios nodos
- Cada nodo tiene un nodo *padre* y guarda un valor
- El nodo *raíz* es aquel que no tiene ningún padre.
- Usualmente se tiene solo una referencia al nodo padre y se puede explorar el árbol a partir de este

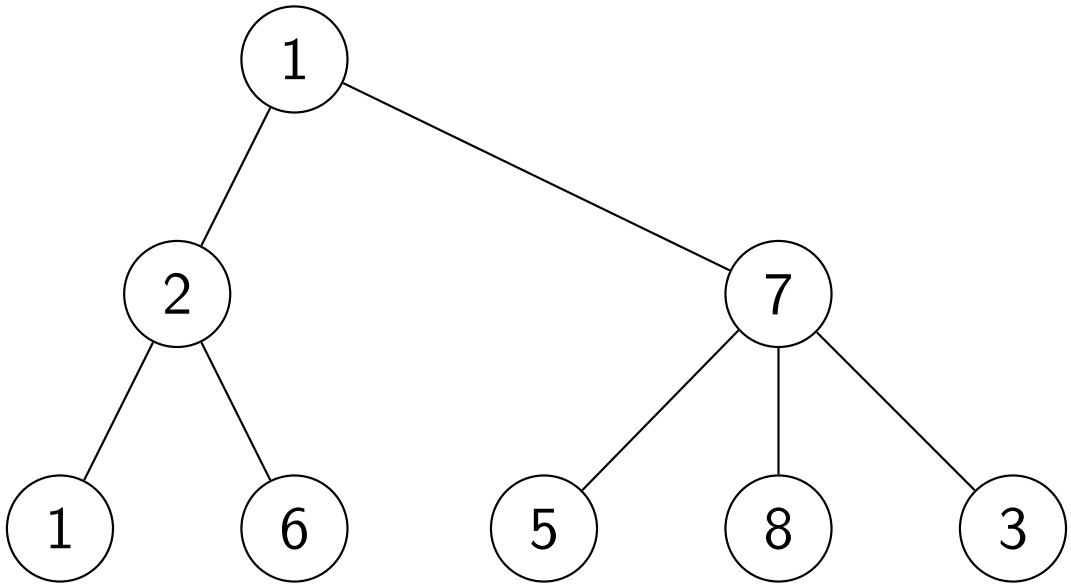
Árboles

- Es una estructura de datos jerárquica, conformado por varios nodos
- Cada nodo tiene un nodo *padre* y guarda un valor
- El nodo *raíz* es aquel que no tiene ningún padre.
- Usualmente se tiene solo una referencia al nodo padre y se puede explorar el árbol a partir de este
- Existen varias formas de implementar un árbol, cada una con sus ventajas y desventajas, tanto en términos de memoria como de tiempo

Árboles

- Es una estructura de datos jerárquica, conformado por varios nodos
- Cada nodo tiene un nodo *padre* y guarda un valor
- El nodo *raíz* es aquel que no tiene ningún padre.
- Usualmente se tiene solo una referencia al nodo padre y se puede explorar el árbol a partir de este
- Existen varias formas de implementar un árbol, cada una con sus ventajas y desventajas, tanto en términos de memoria como de tiempo
- Se conoce como *altura* del árbol al número de niveles que tiene el árbol

Árboles



Árboles

```
#include<cstdlib>

typedef struct tnode{
  struct tnode *parent;
  int data;
}node;

node *NewNode(int d, node *parent) {
  node t;
  t.data = d;
  t.parent = parent;
  return &t;
}

int main() {
  node *root = NewNode(1, NULL);
  node *left = NewNode(2, root);
  NewNode(1, left);
  NewNode(6, left);

  node *right = NewNode(7, root);
```

Árboles

```
#include<cstdlib>
#include<cstdio>

using namespace std;

typedef struct tnode{
  struct tnode *leafs[50];
  int n_leafs; // cantidad de hijos
  int data;
}node;

node *NewNode(int d) {
  node *leaf = (node *)malloc(sizeof(node));
  leaf->data = d;
  leaf->n_leafs = 0;

  return leaf;
}

void AppendNode(node *tree, node *leaf) {
  tree->leafs[tree->n_leafs] = leaf;
```

Árboles binarios

- Son árboles en los que cada nodo tiene a lo mucho dos hijos

Árboles binarios

- Son árboles en los que cada nodo tiene a lo mucho dos hijos
- Se dice que un árbol está completo cuando todos los niveles están llenos, excepto el último

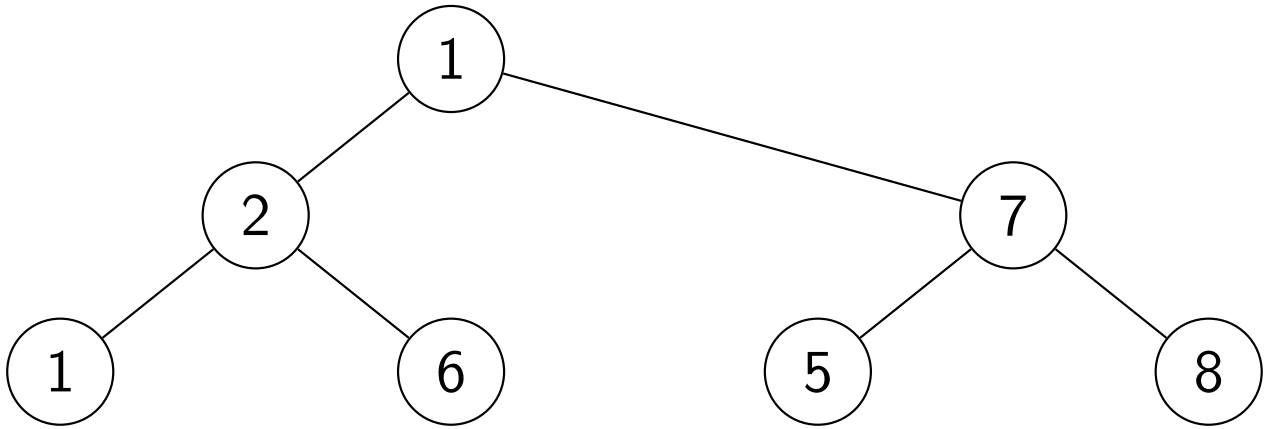
Árboles binarios

- Son árboles en los que cada nodo tiene a lo mucho dos hijos
- Se dice que un árbol está completo cuando todos los niveles están llenos, excepto el último
- Cuando el árbol está completo se cumple la siguiente relación entre la altura y el número de nodos del árbol $h = \lceil \log n \rceil$

Árboles binarios

- Son árboles en los que cada nodo tiene a lo mucho dos hijos
- Se dice que un árbol está completo cuando todos los niveles están llenos, excepto el último
- Cuando el árbol está completo se cumple la siguiente relación entre la altura y el número de nodos del árbol $h = \lceil \log n \rceil$
- Árboles completos pueden ser fácilmente representados con un arreglo; si el árbol no está completo es mejor usar un map

Árboles binarios



Árboles binarios

```
#include<vector>

using namespace std;

int ParentIndex(int n) {
  return (n-1)/2;
}

int main(){
  vector<int> T;

  T.push_back(1);
  T.push_back(2);
  T.push_back(7);
  T.push_back(1);
  T.push_back(6);
  T.push_back(5);
  T.push_back(8);
}
```

Árboles binarios

```
#include<map>

using namespace std;

int ParentIndex(int n) {
  return (n-1)/2;
}

int main(){
  map<int, int> T;

  T[0] = 1;
  T[1] = 1;
  T[2] = 7;
  T[3] = 1;
  T[4] = 6;
  T[5] = 5;
  T[6] = 8;
}
```

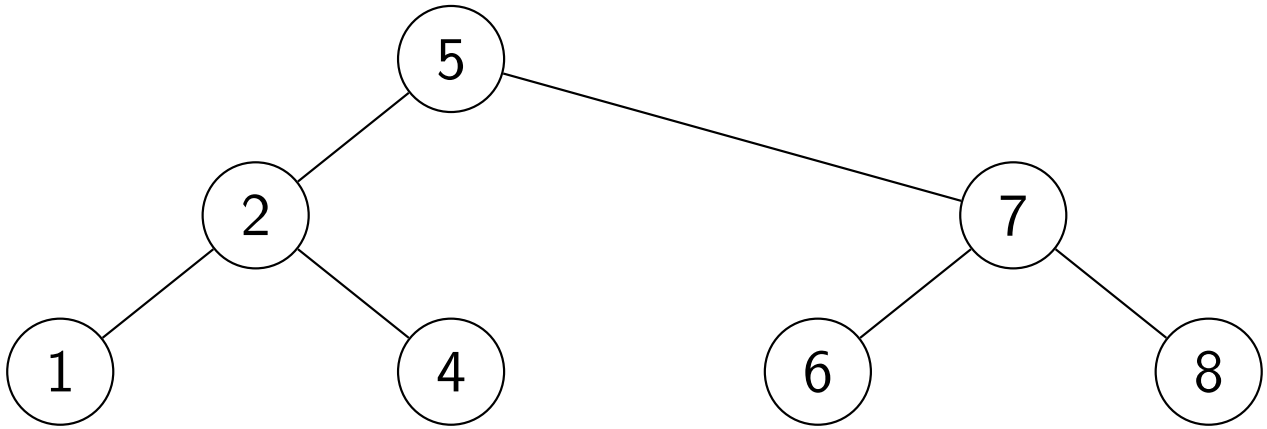
Árboles de búsqueda binarios

- Son árboles binarios tal que para cualquier nodo, el hijo de la izquierda es menor que el nodo padre, y el hijo de la derecha es mayor que el nodo padre

Árboles de búsqueda binarios

- Son árboles binarios tal que para cualquier nodo, el hijo de la izquierda es menor que el nodo padre, y el hijo de la derecha es mayor que el nodo padre
- Se puede generalizar la noción al integrar una relación de orden entre el nodo padre y los nodos hijos

Árboles de búsqueda binarios



Árboles de búsqueda binarios

```
#include<cstdlib>
#include<cstdio>

typedef struct tbst{
  struct tbst *left;
  struct tbst *right;
  int data;
} bst;

bst *NewNode(int n) {
  bst *t = (bst *)malloc(sizeof(bst));
  t->data = n;
  t->left = NULL;
  t->right = NULL;
  return t;
}

// agrega recursivamente el nodo child al nodo root
void AppendNode(bst *root, bst *child) {
  if( child->data < root->data ){
    if ( root->left != NULL ){
```