Estruturas de Dados – STL

Estruturas de Dados - É uma forma de **armazenar e organizar dados**, provendo maneiras eficientes para a realização de inserções, consultas, buscas, atualizações e remoções.

STL (Standard Template Library) - É uma biblioteca de algoritmos e estruturas de dados genéricas, integrada à biblioteca padrão de C++. Podemos ver a STL como uma **caixa de ferramentas** que traz soluções para muitos problemas que envolvem estruturas de dados.

Iremos dividir as estruturas de dados da STL em três categorias:

- **Sequenciais**
  - Vector, Deque.
- **Adaptadores**
  - Pilha, Fila, Fila de Prioridade(Não sequencial).
- **Não Sequenciais**
  - Map, Set, MultiSet.

Sequenciais

Vector – Representam o mesmo tipo de estrutura de estrutura de um vetor, e são manipulados com a mesma eficiência, mas seu tamanho é dinâmico.

- Elementos podem ser acessados normalmente pelo operador [].
- Sua utilização é importante quando não sabemos o tamanho do vetor com antecedência.
- São bastante eficientes no acesso aos elementos e na inserção e remoção de elementos no seu fim.
- É necessário incluir a biblioteca <vector>.
- Muito utilizado em algoritmos de grafos.
- Principais operações:
  - Push_back() – Adiciona um elemento no fim do vector.
  - Size() – Retorna a quantidade de valores do vector.
  - Erase() – Apaga um elemento no vector e organiza dinamicamente os que sobrarem.
  - Clear() – Apaga todos elementos do vector.
  - Pop_back() – Remove o ultimo elemento inserido.
  - É possível utilizar operadores como "**= < >**" entre dois vectors.

```
1.      vector<int> v;
2.
3.      for(int i=1; i<=6; i++)
4.          v.push_back(i);     // Adicionei os elementos de 1 a 6, sempre no final do vector
5.
6.      int tam = v.size();   // pego o tamanho do vector
7.      printf("%d\n",tam);   // 6
8.
9.      for(int i=0; i<tam; i++)
10.         printf("%d ",v[i]);   // 1 2 3 4 5 6
11.
```

```
12.     v.erase(v.begin() + 1); // sintaxe: nome.begin() + quantidade de posições (nesse caso, apago o 2).
13.     tam = v.size(); // 5
14.
15.     for(int i=0; i<tam; i++)
16.         printf("%d ",v[i]);  // 1 3 4 5 6
17.
18.     v.pop_back(); // apago o ultimo elemento
19.     tam = v.size(); // 4
20.
21.     for(int i=0; i<tam; i++)
22.         printf("%d ",v[i]); // 1 3 4 5
23.
24.     v.push_back(2); // adiciono o 2 de volta
25.     tam = v.size(); // 5
26.
27.     for(int i=0; i<tam; i++)
28.         printf("%d ",v[i]); // 1 3 4 5 2
29.
30.     v.clear(); // limpo o vector
31.     tam = v.size();
32.     printf("%d\n",tam); // 0
```

Deque – É uma estrutura bem parecida com o vector, uma diferença notável é que podemos agora acrescentar e remover valores tanto no seu final, quanto no seu início!

- É necessário incluir a biblioteca <deque>
- As operações são as mesmas do vector, porém temos agora:
  - Push_front() – Insere um elemento no seu início.
  - Pop_front() – Excluir seu primeiro elemento.

```
1.      deque<int> d;
2.
3.      for(int i=1; i<=3; i++)
4.          d.push_back(i);  // adiciono 1 2 3 no seu final
5.      for(int i=4; i<=6; i++)
6.          d.push_front(i);  // adiciona 4 5 6 no seu começo
7.
8.      for(int i=0; i<d.size(); i++)
9.          cout<<d[i]<<" ";  // 6 5 4 1 2 3
10.
11.     d.pop_front(); // excluo o valor 6
12.     d.pop_back();  // excluo o valor 3
13.
14.     for(int i=0; i<d.size(); i++)
15.         cout<<d[i]<<" "; // 5 4 1 2
```

Adaptadores

Pilha (Stack) – Projetado especificamente para trabalhar em um contexto LIFO (Last in, First out), onde elementos são sempre inseridos e removidos do topo.

- Último que entra, é o primeiro a sair.
- Não permite acesso aleatório aos elementos.
- É necessário incluir a biblioteca <stack>
- Principais operações:
  - Push() – Insere um elemento no topo da pilha.
  - Pop() – Remove o elemento do topo.
  - Empty() – Checa se a pilha está vazia ou não.
  - Size() – Retorna a quantidade de elementos na pilha.
  - Top() – Acessa o elemento do topo.

```
1.      stack<int> s;
2.
3.      for(int i=1; i<=6; i++)
4.          s.push(i);              // o elemento do topo será o 6, o elemento da base será o 1
5.
6.      printf("%d\n",s.top()); // 6
7.      s.pop();  // Removo o 6
8.      printf("%d\n",s.size()); // Tamanho 5
9.
10.     while(!s.empty())  // Enquanto não estiver vazia
11.         s.pop();        // Removo
12.
13.     printf("%d\n",s.size()); // Tamanho Final 0
```

Fila (Queue) – Projetado especificamente para trabalhar em um contexto FIFO (First in, First out), onde elementos são sempre inseridos no fim e removidos do início.

- Primeiro que entra, é o primeiro a sair.
- É necessário incluir a biblioteca <stack>
- Não permite acesso aleatório aos elementos.
- Usaremos em grafos.
- Principais operações:
  - Push() – Insere um elemento no final da fila.
  - Pop() – Remove o elemento do começo da fila.
  - Empty() – Checa se a fila está vazia ou não.
  - Size() – Retorna a quantidade de elementos na fila.
  - Front() – Acessa o elemento da frente.
  - Back() – Acessa o elemento do final.

```
queue<int> q;

for(int i=1; i<=6; i++)
    q.push(i);          // elemento do inicio será o 1, elemento do final será o 6.

printf("%d\n",q.front());  // 1
printf("%d\n",q.back());  // 6

q.pop();            // Removo o 1
printf("%d\n",q.size());  // Tamanho 5

while(!q.empty())  // Enquanto não estiver vazia
    q.pop();   // Vou removendo

printf("%d\n",q.size()); // Tamanho Final 0
```

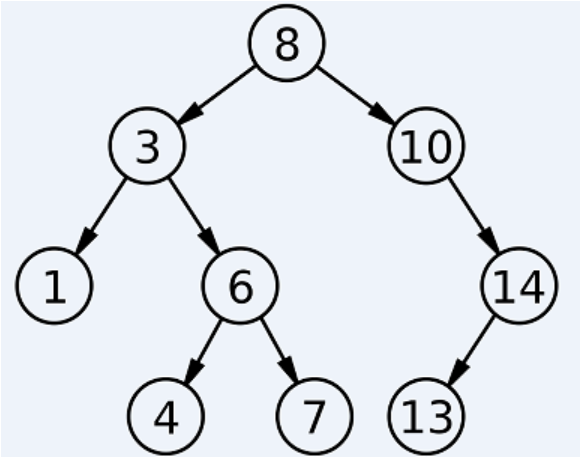
Fila de Prioridade (Priority_Queue) – Filas de prioridade são um tipo de adaptador projetado especificamente para que seu primeiro elemento seja sempre o maior entre todos os elementos.

- É necessário incluir a biblioteca <queue>
- O elemento do topo é sempre o maior, independente da ordem inserida.
- Usaremos em grafos.
- Possui as mesmas operações de uma fila normal.

```
priority_queue<int> pq;

pq.push(6);  pq.push(2);
pq.push(10); pq.push(3);  // inserimos elementos aleatórios

printf("%d\n",pq.top());  // independente da ordem de inserção, o topo será o maior, 10
pq.pop(); // removi o 10
printf("%d\n",pq.top()); // o novo topo é 6

pq.push(100); // inserindo 100
pq.push(1);  // inserindo 1
printf("%d\n",pq.top()); //o novo topo é 100
printf("%d\n",pq.size()); // tamanho = 5

while(!pq.empty()){ // enquanto não estiver vazia
    printf("%d ",pq.top());  // 100 6 3 2 1
    pq.pop(); // removo o topo
}
```

Não Sequenciais

Os elementos não são guardados de forma sequencial na estrutura. Usando essas estruturas nos problemas corretos, podemos fazer consultas de formas rápidas e acelerar nossos algoritmos. São implementadas em forma de árvore binária de busca. Este tipo de árvore permite inserção, busca e remoção em O(logN).



Mapa(Map) – Diferente de um vetor normal, podemos ter nossa chave(índice) e qualquer tipo! Para percorrermos seus elementos, visto que não são armazenados de forma sequencial, devemos criar um iterador(ponteiro inteligente da STL).

- Bastante usado em diversos tipos de problemas.
- Os elementos inseridos são ordenados automaticamente (menor par maior).
- Não permite chaves repetidas. Incluir biblioteca Map.
- Principais operações.
  - Clear(). – Limpa todos os elementos do mapa.
  - Size() - Retorna a quantidade de elementos no mapa.

```
1.      map<string, double> mp;   // Chave do tipo String e conteúdo do tipo double
2.      map<string, double> :: iterator it; // Iterador para pecorrero mapa, deve ser do mesmo tipo.
3.
4.      mp["abacaxi"] = 5.99;  // chave = abacaxi, conteúdo = 5.99
5.      mp["notebook"] = 2000.50;
6.      mp["bolo"] = 3.30;
7.      mp["abacaxi"] += 1.00;  // posso somar 1.00 no conteúdo do abacaxi
8.      mp["bolo"] = 4.40;  // não irá inserir outro bolo, e sim substituir o valor do antigo
9.
10.     printf("%d\n",mp.size()); // 3 elementos
11.
12.     for(it = mp.begin(); it != mp.end(); it++) // percorro o mapa usando o iterador it que criei
13.         cout<<it->first<<" "<<it->second<<endl // mostro a chave(first) e conteúdo(second)
14.                                             // abacaxi 6.99    bolo 4.40     notebook 2000.50
15.     mp.clear(); // limpo o mapa
```

Set – Seus dados são organizados da mesma forma que um map. Porém não formam “Pares” de (chave – conteúdo), e sim apenas um valor inserido no set.

- Os elementos inseridos são ordenados automaticamente (menor par maior).
- Não permite valores repetidos (são ignorados).
- Devemos também criar um iterador para percorrer um Set.
- Incluir biblioteca <set>.
- Principais operações.
  - Insert() – Insere um elemento no Set.
  - Size() – Retorna a quantidade de elementos **distintos**.
  - Erase() – Apaga um valor especifico.

```
1.      set<int> s;
2.      set<int> :: iterator it;   // crio o iterador para percorrer o set depois, mesmo tipo.
3.
4.      s.insert(10);
5.      s.insert(8);
6.      s.insert(9);
7.      s.insert(9);  // segundo nove não será inserido.
8.
9.      for(it = s.begin(); it != s.end(); it++) // percorro o set e exibo os valores
10.         cout<< *it <<" ";    //  8 9 10     Lembrando que o set ordena automaticamente
11.
12.     s.erase(9);  // apago o 9 do set
13.
14.     for(it = s.begin(); it != s.end(); it++)
15.         cout<< *it <<" ";    //  8 10
16.
17.     s.insert(1); s.insert(15);  // insiro 1 e 15... set atual   1 8 10 15
18.
19.     it = s.end(); it--;  // acessando o ultimo elemento. Lembrar de dar um it--
20.     cout<<*it<<endl;  // 15
21.
22.     it = s.begin(); // primeiro elemento
23.     cout<<*it<<endl; // 1
24.
25.     s.clear(); // Limpo o set
```

Multiset – Idêntico ao Set, porém permite trabalhar com elementos repetidos.

Pair

Algumas vezes precisamos trabalhar com pares de valores, a biblioteca utility nos fornece o **pair**, facilitando a nossa vida na hora de codificar.

Podemos e iremos usar pair com todas as estruturas da STL visto acima.

Para formarmos os pares usaremos o comando make_pair(X,Y);

Acessaremos sempre seu primeiro elemento usando first e o segundo elemento usando second.

Exemplo:

```
1.      vector<pair<int, int> > v;   // crio um vector de par de inteiros
2.
3.      v.push_back(make_pair(2,20));  // adiciono o par (2,20)
4.      v.push_back(make_pair(1,10));
5.      v.push_back(make_pair(5,500));
6.
7.      int tam = v.size();  // tamanho = 3,   3 pares adicionados
8.
9.      for(int i=0; i<tam; i++)  // percorro o vector, posso usar iterador também
10.         cout<<v[i].first<<"  -  "<<v[i].second<<endl;   // 2 - 20    1 - 10     5 - 500
11.
12.     sort(v.begin(), v.end());  // para ordenar um vector com sort, a sintaxe eh essa.
13.                                // por padrão ele ordena em relação ao primeiro valor do par (first)
14.                               // mas podemos criar uma função de comparação p ordenar de qualquer jeito..
15.     for(int i=0; i<tam; i++)
16.         cout<<v[i].first<<"  --  "<<v[i].second<<endl;  // 1 - 10    2 - 20    5 - 500
```