Um exemplo apropriado do uso de uma estrutura dinâmica e flexível de dados é o processo de ordenação topológica. Esse processo consiste na ordenação de elementos, no qual é definida uma ordenação parcial, isto é, no qual uma ordenação é efetuada somente sobre alguns pares de elementos e não sobre todo o seu conjunto.

Existem vários exemplos de conjuntos com ordens parciais, dentre eles:

1. Em projetos de engenharia ou manufatura, uma tarefa é dividida em subtarefas. Em geral, o término de certas subtarefas deve preceder a execução de outras subtarefas. Se uma subtarefa x deve preceder uma subtarefa y, isto será denotado por x < y. A ordenação topológica, neste caso, consiste em dispor as subtarefas em uma ordem tal que, antes da iniciação de cada subtarefa, todas as subtarefas de que ela depende tenham sido previamente completadas;
2. Em um currículo universitário, algumas disciplinas devem ser cursadas antes de outras, uma vez que se baseiam nos tópicos apresentados nas disciplinas que são seus pré-requisitos. Se uma disciplina x é pré-requisito da disciplina y, a notação será x < y. A ordenação topológica corresponde, no caso, a arranjar as disciplinas em uma ordem tal que nenhuma delas exija como pré-requisito outra que não tenha sido previamente cursada.
3. Em um programa, alguns procedimentos podem conter referências (chamadas) a outros procedimentos. Se um procedimento x é chamado por um procedimento y, denota-se o fato por x < y. Neste caso, a ordenação topológica implica o arranjo das declarações de procedimento em tal forma que nunca haja referências a procedimentos a serem declarados posteriormente.

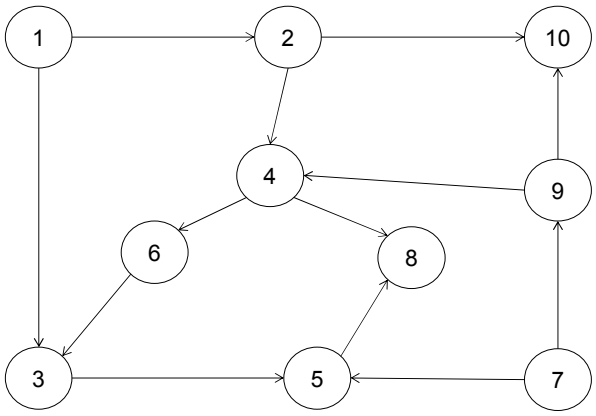
Em geral, uma ordem parcial de um conjunto *S* é uma relação entre os elementos de *S*. Esta relação, denotada pelo símbolo <, verbalmente lida como "precede", deve satisfazer as três seguintes propriedades (axiomas), para quaisquer elementos distintos *x*, *y*, *z* de *S*:

(a) se $x < y$ e $y < z$ então $x < z$ (transitividade)
(b) se $x < y$ então não ocorre $y < x$ (assimetria)
(c) não ocorre $z < z$ (não-reflexivo)

Admite-se, por motivos óbvios, que o conjunto S é finito. Uma ordem parcial pode ser representada por um diagrama (ou grafo) no qual os vértices (ou nós) denotam os elementos de *S* e as setas (arestas) representam as relações de ordem. Um exemplo de diagrama é mostrado na Figura 1(ii) e do respectivo arquivo de entrada contendo a relação S na Figura 1(i):

```
Projeto Diagrama Exemplo 1
1 < 2
2 < 4
4 < 6
2 < 10
4 < 8
6 < 3
1 < 3
3 < 5
5 < 8
7 < 5
7 < 9
9 < 4
9 < 10
```

(i)



(ii)

Figura 1: Um exemplo de dados de entrada (i) e o diagrama correspondente (ii)

O problema da ordenação topológica consiste em encontrar uma ordem linear a partir de uma ordem parcial. Graficamente, isto resulta em um arranjo linear dos nós do diagrama, de tal maneira que todas as setas

tenham o mesmo sentido (para a direita, por exemplo), conforme a Figura 2. As propriedades (a) e (b) da ordem parcial garantem a ausência de ciclos no diagrama o que permite transformar uma ordem parcial em uma ordem linear dos elementos.

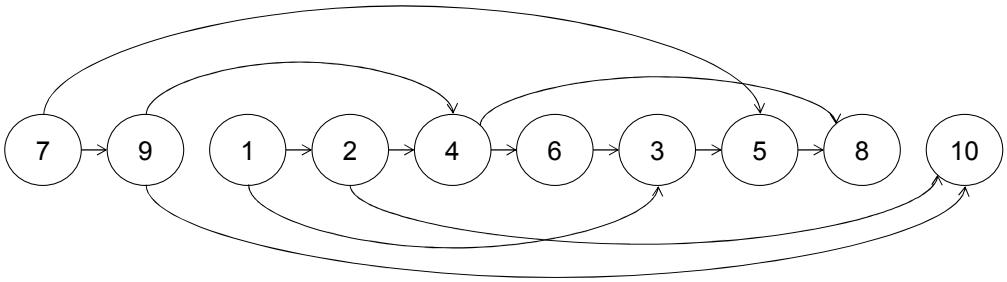


Figura 2: Um exemplo de solução do problema da Figura 1

O algoritmo para encontrar uma das possíveis ordenações lineares é bastante simples:

1) Ler a ordem parcial da entrada E, armazenando S, para cada nó:
    - O número (quantidade) de seus predecessores
    - A lista de seus sucessores
2) Escolher os nós que não são precedidos por quaisquer outros denotando esse conjunto de nós sem predecessores por *P*
    - É necessário que haja, no mínimo, um elemento nessas condições; caso contrário existiria um ciclo e o conjunto não seria parcialmente ordenado
3) Escolher um nó *x* de *P*, colocando *x* no início da lista de saída *L*; em seguida o nó *x* deve ser eliminado do conjunto *S*
4) O conjunto *S* resultante ainda se encontra parcialmente ordenado, devendo então o mesmo algoritmo ser aplicado sucessivamente até que o conjunto *S* se esgote

Ao final do algoritmo, se a lista *L* contém o mesmo número de nós no grafo então uma ordem parcial foi encontrada e seu resultado encontra-se na própria lista *L*; caso contrário, o grafo não constitui uma ordem parcial (há ciclos). Um exemplo desta última situação é mostrada na Figura 3.

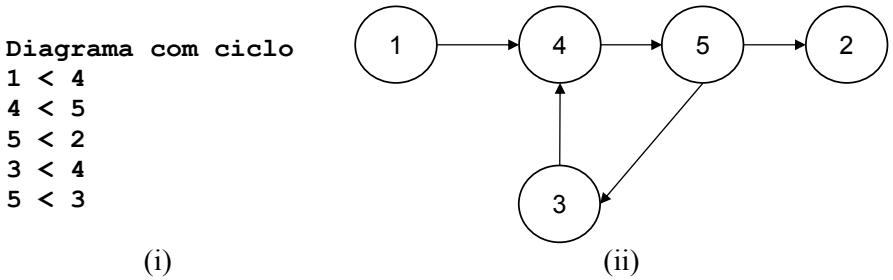


Figura 3: Um exemplo de dados de entrada (i) e o diagrama contendo um ciclo (ii)

Ao escrever o código, tenha em mente que:

- Pode haver mais de um nó sem arestas de entrada
- Pode haver mais de um nó sem arestas de saída

Observe que o programa não precisa verificar por essas condições explicitamente, pois se sua implementação da solução apresentada estiver correta, ela gerenciará todos esses casos naturalmente.

Para se poder descrever detalhadamente este algoritmo, deve-se escolher uma estrutura de dados para representar o conjunto S, bem como a representação de suas ordenações, cuja implementação será realizada por meio da classe *TopSort* (os métodos em negrito são aqueles que você deve implementar para completar com sucesso este laboratório; os demais métodos já se encontram prontos para uso e não necessitam ser alterados). A escolha desta representação é determinada pelas operações a serem realizadas, em particular a operação de seleção de elementos sem predecessores.

```
class TopSort
{ public:
    TopSort(string projectName = "Unnamed project");
```

```
    ~TopSort();
    int GetNodes();
    int GetEdges();
    string GetProjectName();
    void Clear();
    void AddRelation(TaskType predecessor, TaskType sucessor);
    List<TaskType> FindTopSort();

  private:
    // definicao de tipos
    struct leader;   // declaracoes incompletas, pois leader e trailer sao dependentes entre si
    struct trailer;  // declaracoes completas mais abaixo no codigo

    typedef leader  * leaderPointer;
    typedef trailer * trailerPointer;

    struct leader
    { TaskType task;
      int      predecessors;       // numero de precedessores da tarefa task
      leaderPointer  nextLeader;
      trailerPointer nextTrailer;
    };

    struct trailer
    { leaderPointer  nextLeader;
      trailerPointer nextTrailer;
    };

    // definicao de campos do objeto
    leaderPointer head, sentinel; // inicio e final da estrutura de dados
    string projectName;           // nome do projeto (grafo)
    int nodes;                    // numero de elementos no grafo
    int edges;                    // numero de arestas no grafo

    // definicao de metodos privados
    leaderPointer InsertNode(TaskType task);
    void InsertEdge(leaderPointer &p, leaderPointer &q);
};
```

Dado que o número *n* de elementos de *S* não é conhecido a priori, o conjunto pode ser organizado convenientemente na forma de uma lista ligada. Em consequência, na descrição de cada elemento deverá constar uma informação adicional, correspondente ao apontador para o próximo elemento da lista. As chaves serão consideradas como números inteiros (mas não necessariamente inteiros consecutivos de 1 a *n*). Analogamente, o conjunto de cada sucessor dos elementos é representado, muito adequadamente, por uma lista ligada. Cada elemento da lista de sucessores é descrito por uma identificação e por um apontador para o próximo elemento desta lista. Aos descritores da lista principal, em que cada elemento de S ocorre exatamente uma vez, será dado o nome de preâmbulos (*leaders*) e aos descritores dos elementos da cadeia de sucessores, o nome de postâmbulos (*trailers*), conforme mostrado na Figura 4.

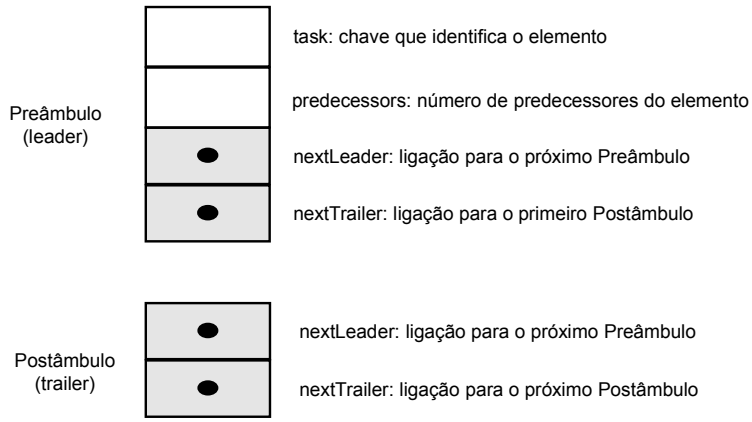


Figura 4: Dois tipos de nós para solução do problema

Admita que o conjunto S e suas relações de ordenação sejam inicialmente representados por uma sequência de pares de nós, como já mostrado na Figura 1(i), que podem ser implementados segundo o código fornecido na Figura 5(i) e saída correspondente na Figura 5(ii).

```
int main()
{
    TopSort T("Projeto Diagrama Exemplo 1");

    T.AddRelation(1,2);
    T.AddRelation(2,4);
    T.AddRelation(4,6);
    T.AddRelation(2,10);
    T.AddRelation(4,8);
    T.AddRelation(6,3);
    T.AddRelation(1,3);
    T.AddRelation(3,5);
    T.AddRelation(5,8);
    T.AddRelation(7,5);
    T.AddRelation(7,9);
    T.AddRelation(9,4);
    T.AddRelation(9,10);

    List<TaskType> L = T.FindTopSort();
    cout << T.GetNodes() << " nodes, "
         << T.GetEdges() << " edges" << endl;
    if(L.Size() == T.GetNodes())
      cout << "Ordem parcial: " << L.toString() << endl;
    else
      cout << "Nao eh ordem parcial" << endl;
}
```

```
10 nodes, 13 edges
Ordem parcial: 7,9,1,2,4,6,3,5,8,10
```

(i) (ii)

Figura 5: Entrada (i) e solução (ii) do diagrama da Figura 1

```
int main()
{
    TopSort T("Diagrama com ciclo");

    T.AddRelation(1,4);
    T.AddRelation(4,5);
    T.AddRelation(5,2);
    T.AddRelation(3,4);
    T.AddRelation(5,3);

    List<TaskType> L = T.FindTopSort();
    cout << T.GetNodes() << " nodes, "
         << T.GetEdges() << " edges" << endl;
    if(L.Size() == T.GetNodes())
      cout << "Ordem parcial: " << L.toString() << endl;
    else
      cout << "Nao eh ordem parcial" << endl;    }
```

```
5 nodes, 5 edges
Nao eh ordem parcial
```

(i) (ii)

Figura 6: Entrada (i) e solução (ii) do diagrama da Figura 3

A primeira parte do programa de ordenação topológica deve inserir os dados de entrada em uma estrutura de lista (com sentinela), inicialmente mostrada na Figura 7.

Figura 7: Estrutura de dados inicial, elemento sentinela ao final.

A inserção de dados é feita por meio de chamadas sucessivas ao método *AddRelation(x, y)* de pares de chaves (tarefas) x e y ($x < y$).

```
void TopSort::AddRelation(TaskType x, TaskType y)
{ leaderPointer p,q;

  p = InsertNode(x);
  q = InsertNode(y);
  InsertEdge(p,q);
}
```

Portanto p e q são ponteiros que indicam a posição inicial das representações de x e y na lista ligada. Estes nós devem ser localizados por meio da aplicação de uma operação de busca e, caso não tenham sido encontrados na lista, devem nela ser inseridos. Essa tarefa é muito frequente ao se trabalhar listas encadeadas e é conhecida como "busca com inserção", conforme já mencionado em aula. Essa tarefa deve ser

implementada pelo método *InsertNode(TaskType x)* da classe TopSort em C++. Este método (função) deve retornar no nome da função um ponteiro para o preâmbulo contendo o elemento *x*, conforme Figura 8.

Figura 8: Estrutura após comando *p = InsertNode(1);*

O mesmo processo de busca com inserção é efetuado para o elemento y, resultando na estrutura mostrada na Figura 9.

Figura 9: Estrutura após comando *q = InsertNode(2);*

A seguir, um novo elemento, acompanhado da identificação de *y*, é adicionado a lista que sucede o elemento *x* e a contagem dos predecessores de *y* é incrementada de uma unidade, pelo método *InsertEdge(p,q)*, conforme Figura 10.

Figura 10: Estrutura após comando *InsertEdge(p,q);*

Esta passagem é denominada de fase de entrada (1). Na Figura 11 encontra-se ilustrada a estrutura de dados completa que deve ser gerada após todas as chamadas ao método *AddRelation(x,y)* para o diagrama da Figura 1 e código C++ correspondente na Figura 5(i).

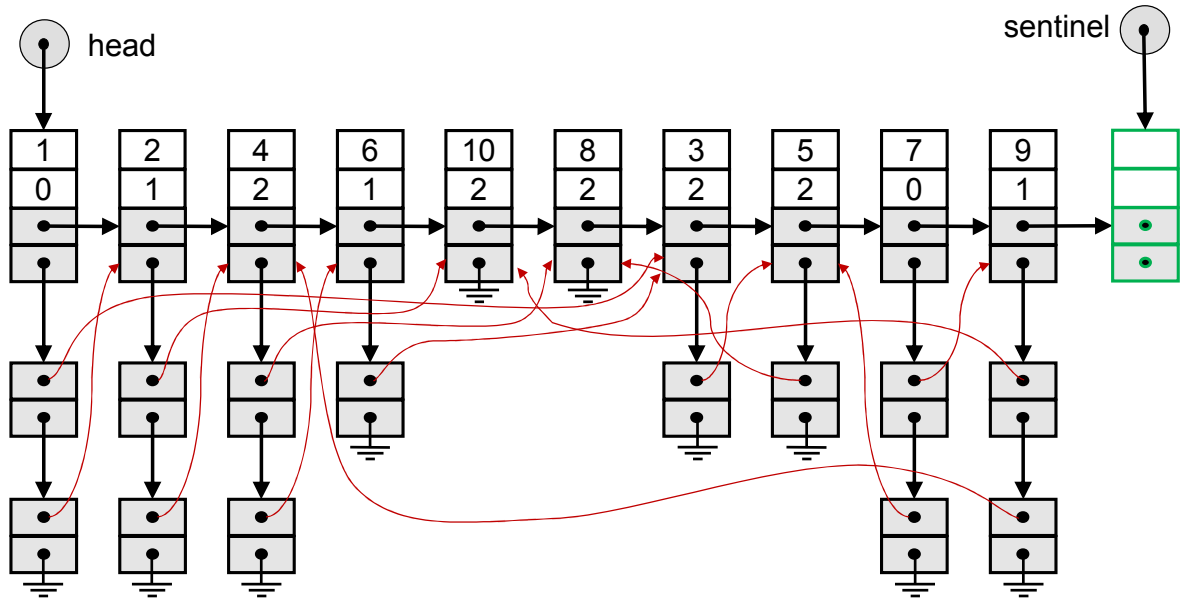


Figura 11: Estrutura de lista gerada pela classe TopSort

Após a construção da estrutura de dados da Figura 11 na fase de entrada, o processo real de ordenação topológica pode ser efetuado conforme foi descrito acima no algoritmo. Porém, a cada passagem devem ser selecionados, repetidamente, os elementos sem predecessores. Assim, é computacionalmente interessante coletar, de início, todos estes elementos em uma lista ligada. Considerando que a lista original de preâmbulos não será mais necessária, o mesmo campo denominado **`nextLeader`** poderá ser reutilizado, agora com a função de interligar os elementos cujos preâmbulos indicam zero predecessores. Esta operação de substituição de uma lista por outra ocorre com frequência em programas que envolvem o processamento de listas e é expressa detalhadamente no fragmento de código seguinte. Por eficiência, esta operação constrói a nova lista em ordem reversa. Sejam *p* e *q* ponteiros para preâmbulos:

```
// busca por preambulos sem predecessores
p = head;
head = NULL;
while(p != sentinel)
{ q = p;
  p = q->nextLeader;
  if(q->count == 0) // inserir q na nova lista
  { q->nextLeader = head;
    head = q;
  }
}
```

Considerando a Figura 11, nota-se que a nova lista de preâmbulos é substituída pela da Figura 12, na qual os ponteiros não mostrados permanecem inalterados. Porém, a utilização desta estratégia exigirá atenção especial na programação do método *Clear()* que libera todo espaço alocado, exceto o elemento sentinela.

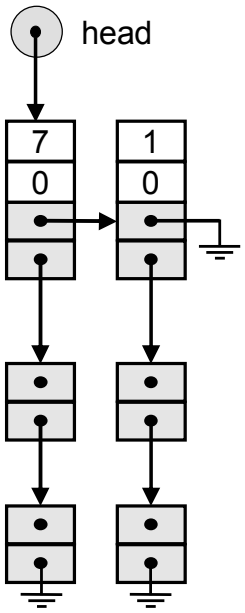


Figura 12: Nova lista de preâmbulos sem predecessores

Após a escolha desta representação para o conjunto parcialmente ordenado S, pode-se finalmente prosseguir com a tarefa de ordenação topológica propriamente dita, isto é, a de gerar a lista de saída. Em uma versão rudimentar isto pode ser efetuado da seguinte forma (*t* é um ponteiro para postâmbulo):

```
q = head;
while (q != NULL) // efetuar a saída deste elemento, depois eliminá-lo
{ colocar q->key na lista de saída L
  n = n - 1;        // um elemento a menos em S
  t = q->nextTrailer;
  q = q->nextLeader;
  decrementar o contador do predecessor de todos os seus sucessores
  na lista de postâmbulos t; se algum contador se tornar 0, inserir este
  elemento na lista q de preâmbulos.
}
```

Note-se que foi introduzido um contador *n* para contabilizar os preâmbulos gerados na fase de entrada. Este contador é decrementado cada vez que um elemento preâmbulo é inserido na sequência de saída, durante a fase de saída. Deve, portanto, anular-se no final da execução do programa. Caso isto não ocorra, é uma constatação de que restaram elementos na estrutura de modo tal que nenhum deles deixa de apresentar um predecessor. Neste caso, é evidente que o conjunto *S* não apresenta a propriedade da ordenação parcial. A fase de saída, acima programada, é um exemplo de um processo em que é mantida uma lista que se expande e contrai, isto é, cujos elementos são inseridos e removidos em uma ordem predeterminada. Trata-se, portanto, de um exemplo de um processo que utiliza a total flexibilidade que a lista ligada explícita oferece.

Você pode incluir quaisquer subalgoritmos (funções, procedimentos ou métodos) ou campos no objeto *TopSort* que se fizerem necessários, porém não remova ou altere os métodos e campos já fornecidos (caso contrário, é possível que a plataforma CxxTest atribua pontuação menor que o máximo admissível ao seu trabalho, mesmo que ele esteja correto).

Submissão

Submeta sua implementação no sistema Web-CAT, disponível em http://kode.ffclrp.usp.br:8080/WebCat. Este laboratório deve ser submetido individualmente. Antes de submeter os arquivos necessários, coloque seu nome completo em todos os arquivos sendo submetidos, na forma de comentário no início de cada arquivo (.h ou .cpp).

Compacte os seguintes arquivos em um único arquivo .zip (não utilize espaços no nome do arquivo compactado, nem adicione pastas/diretórios no arquivo compactado):

- ListTemplate.h (interface e implementação da classe *List* usando *Templates*)
- TopSort.h, TopSort.cpp (interface e implementação completa da classe *TopSort*)
- StudentEmptyTest.h (arquivo de teste do aluno, no formato da plataforma CxxTest)

Não inclua o programa principal, ou seja a função *main*(), na submissão ao Web-CAT. Respeite os nomes de arquivos, da classe e dos métodos. Submeta o arquivo compactado ao Web-CAT. Em caso de dúvida, procure o professor.

Avaliação

Na nota do trabalho também serão considerados os seguintes critérios (além dos já mencionados nas Disposições Gerais entregues no início do semestre):

- **Correção:** O programa faz o que foi solicitado? Faz tudo o que foi solicitado? Utiliza encapsulamento de informação? (i.e., acessa adequadamente os ADTs definidos?) Não há vazamento de memória?
- **Eficiência: A**s operações são executadas da maneira mais eficiente para cada estrutura de dados? Evita código duplicado/redundante/não atingível?
- **Interface:** É simples de usar, genérico, prático, tolera os erros mais óbvios? O trabalho foi entregue dentro das especificações (um arquivo *.h* para cada *.cpp* implementando um ADT? Os arquivos estão em formato ZIP, com os nomes de arquivos solicitados)?
    - interface do programa;
    - implementação dos métodos;
- **Código fonte:** é claro (*layout*, espaçamento, organização em geral), nomes de variáveis são sugestivos, e há documentação/comentários apropriados no código? Faz uso de pré- e pós-condições? Quando aplicável, faz uso de sublagoritmos (funções, procedimentos ou métodos) adicionais que melhoram a legibilidade do(s) método(s) solicitado(s) sem comprometer sua eficiência (por exemplo, na notação assintótica **O(*n*),** onde ***n*** representa o tamanho da entrada?)